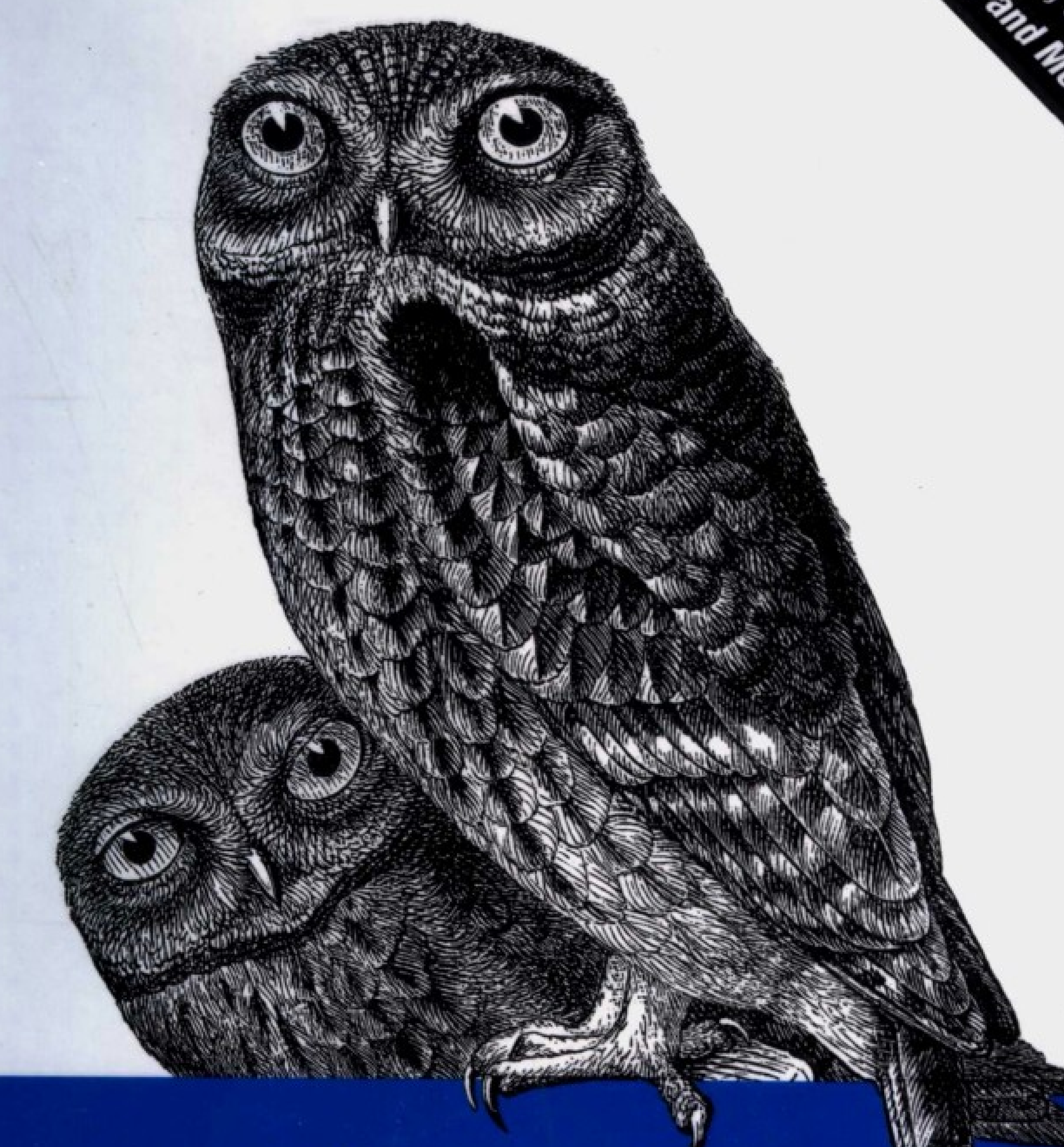
Mastering Regular Expressions

Understand Your Data and Be More Productive

第3版

For Perl, PHP, Java, .NET, Ruby, and More!



精通 正则表达式

Jeffrey E.F. Friedl 著

余晟 译

O'REILLY®


電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

精通正则表达式（第3版）

Mastering Regular Expressions, 3rd Edition



[美] Jeffrey E.F. Friedl 著

余晟 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是在线出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为20世纪最重要的50本书之一)到GNN(最早的Internet 门户和商业网站)，再到 WebSite（第一个桌面 PC 的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 **|** 软考历年真是解析与答案

软考视频 **|** 考试机构 **|** 考试时间安排

Java 一览无余：**Java** 视频教程 **| Java SE | Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书**+**视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料**(**电子书**+**视频教程**)**下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 **| SQL Server** 篇 **| Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具**+**电子书**+**视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集**(**电子书**+**视频教程**) Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料**<**电子书**+**视频**>**分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍**+**视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

推荐序

一夫当关

IT 产业新技术日新月异，令人目不暇接，然而在这其中，真正能称得上伟大的东西却寥寥无几。1998 年，被誉为“软件世界的爱迪生”，发明了 BSD、TCP/IP、csh、vi 和 NFS 的 SUN 首席科学家 Bill Joy 曾经不无调侃地说，在计算机体系结构领域里，缓存是唯一能称得上伟大的思想，其他的一切发明和技术不过是在不同场景下应用这一思想而已。在计算机软件领域里，情形也大体相似。如果罗列这个领域中的伟大发明，我相信绝不会超过二十项。在这个名单当中，当然应该包括分组交换网络、Web、Lisp、哈希算法、UNIX、编译技术、关系模型、面向对象、XML 这些大名鼎鼎的家伙，而正则表达式也绝对不应该被漏掉。正则表达式具有伟大技术发明的一切特点，它简单、优美、功能强大、妙用无穷。对于很多实际工作来讲，正则表达式简直是灵丹妙药，能够成百倍地提高开发效率和程序质量。CSDN 的创始人蒋涛先生在早年开发专业软件产品时，就曾经体验过这一工具的巨大威力，并且一直印象深刻。而我的一位从事网络编辑工作的朋友，最近也领略了正则表达式的威力——他用 Perl 开发了一个不足 20 行的小程序，使用正则表达式将一项原本每天耗用 10 人时的工作在一分钟之内自动完成。而正则表达式在生物信息学和人类基因图谱的研究中所发挥的关键作用，更是被传为佳话。无论对于软件开发者，还是从事其他知识工作的专业人士，正则表达式都是最有利的工具之一。

所谓正则表达式，就是一种描述字符串结构模式的形式化表达方法。在发展的初期，这套方法仅限于描述正则文本，故此得名“正则表达式（regular expression）”。随着正则表达式研究的深入和发展，特别是 Perl 语言的实践和探索，正则表达式的能力已经大大突破了传统的、数学上的限制，成为威力巨大的实用工具，在几乎所有主流语言中获得支持。为什么正则表达式具有如此巨大的魅力？一方面，因为正则表达式处理的对象是字符串，或者抽象地说，是一个对象序列，而这恰恰是当今计算机体系的本质数据结构，我们围绕计算机所做的大多数工作，都归结为在这个序列上的操作，因此，正则表达式用途广阔。另一方面，与大多数其他技术不同，正则表达式具有超强的结构描述能力，而在计算机中，正是不同的结构把无差别的字节组织成千差万别的软件对象，再组合成为无所不能的软件系统，因此，描述了结构，就等于描述了系统。在这方面，正则表达式的地位是独特的。正因为这两点，在现在的软件开发和日常数据处理工作中，正则表达式已经成为必不可少的工具。如果一个开发工具不支持正则表达式，那它就会被视为玩具语言，如果一个编辑器

不支持正则表达式，那它就会被成为阳春应用。连人们原本并不指望应用正则表达式的商用数据库，各家厂商也竞相以支持正则表达式为卖点。正则表达式的声势之隆，是毋庸置疑的。

非常奇怪的是，这样一个了不起的技术，在我国却并没有得到充分推广。以其价值而言，正则表达式不但值得每一个专业程序员掌握，而且值得所有知识工作者去了解。然而现实情况是，不但一般知识工作者大多闻所未闻，很多专业程序员也视之为畏途。为什么会出现这种情况呢？原因有二。其一，正则表达式产生和发展在 UNIX 文化体系之中，而我国软件开发社群的知识结构长期受到微软的决定，UNIX 文化影响甚微。在 2002 年推出.NET 平台之前，微软在其各项主流平台、产品与开发工具当中，均未对正则表达式给予足够的重视，相应地，我们的开发者们对正则表达式也就知之不多。第二，也是更重要的原因，就是正则表达式并不是那么好掌握的，在通向驾驭正则表达式强大力量的道路上，还是有那么几只拦路虎的，而要打虎过岗，不但要花点功夫，还要有正确的方法。

学习正则表达式，入门不难，看一些例子，试着模仿模仿，就可以粗通，并且在工作中解决不少问题。然而大部分学习者也就就此止步，他们对自己说："正则表达式不过如此，我就学到这里了，以后现用现学就行了。"他们以为自己可以像学习其他技术一样，在实践中逐渐提高正则表达式的应用水平。然而事实上，正则表达式并不是每天都会用到，而其密码般的形象，随着时间的推移很容易被忘记，所以经常发生的情况是，开发者对于正则表达式的记忆迅速消褪，每次遇到新的问题，都要查资料，重新唤回记忆，对于稍微复杂一点的问题，只好求助于现成的解决方案。反反复复，长期如此，不但应用水平难以明显提升，而且会对这项技术逐渐产生一定的恐惧感和厌烦情绪。这还只是应用阶段，正则表达式应用的高级阶段，要求开发者还必须充分理解正则表达式的能力范围，能够将一些正则表达式技术组合应用，达成超乎一般想像的效果。为了高效、正确地解决实际问题，有的时候甚至要求深入理解正则表达式的原理，甚至对于如何实现正则表达式引擎都要有所了解，在此基础上，规避陷阱，优化设计，提高程序执行效率。要达到这样的程度，不经过系统的学习是不可能的。

系统学习正则表达式并不是一件容易的事情，仅仅通过阅读一些"HOW TO"的快餐式文章是不行的，必须有更完整、更系统的资料指导学习。如果你在国外技术社区里询问如何才能系统学习正则表达式，几乎所有的领域专家都会向你推荐一本书——Jeffrey Friedl 的《精通正则表达式》，也就是本书。

这本《精通正则表达式》是系统学习正则表达式的唯一最权威著作。可以说，在今天，如果想理解和掌握正则表达式，想要建立关于这一技术的完整概念体系，想充分发挥其巨大能量，这本书几乎是无法绕开的必经之路。甚至可以说，如果你没有读过这本书，那么你

对于正则表达式的理解和应用能力一定达不到升堂入室的程度。本书第 1 版出版于十年之前，自那时起它就成为正则表达式领域最全面、最受欢迎的代表著作，数以万计的读者通过这本书掌握了正则表达式，成为行家里手。在任何时候，任何地方，只要提到正则表达式著作，人们都会提到这本书。这本书的质量之高，声誉之盛，使得几乎没有人企图挑战它的地位，从而在正则表达式图书领域形成独特的"一夫当关"的局面，称其为正则表达式圣经，绝对当之无愧。

为什么这本书能够表现得如此出色？我认为这其中有三个原因。其一，作者本人具有多年程序开发经验，理论基础深厚，实战经验丰富，对正则表达式这个主题透彻理解，因此在技术上得心应手，底气十足，对于技术上的难点不回避、不含糊。作者高超的技术水平是本书质量的强大保证。其二，作者思路对头，素材组织得当，用例丰富。正则表达式根植于数学理论，却又能在日常俗事上发挥巨大的效用。写这种类型的技术，思路稍微一偏差，就可能走歪路，不是太理论，就是太琐碎，不是太枯燥，就是太浅薄，实在很难把握。作者清楚地认识到，这本书的读者不是计算机科学家，但也不是满足于"知其然而不知其所以然"的快餐式代码小子，而是具有一定理论素养，却又始终以实践为本的专业开发者。他们需要的是面向实践的理论和思想，是实实在在的实战能力，只有满足这种需要，才能够真正打动读者。通读此书，可以说作者对这一路线的把握十分成功，保证了内容大方向的正确。其三，这本书的写法独具匠心，堪称典范。技术图书的主要使命是传播专业知识。而专业知识分为框架性知识和具体知识。框架性知识需要通过系统的阅读和学习掌握，而大量的具体知识，则主要通过日常工作的积累以及随用随查的的学习来逐渐填充起来。本书前六章，以顺序式记述的方式，将正则表达式的系统知识娓娓道来，读者像看故事书似的就建立起整个正则表达式的基本知识体系。而后面的内容，则是方便实际开发中频发查阅之用，包括各大主流语言对正则表达式的支持细节，包含有大量案例。这样的写法，完全符合一般人学习的特点，因此书读起来非常惬意，非常有趣，用的时候查起来又非常方便。这样的著述风格，实在值得学习。

读者可以在没有任何正则表达式的基础上开始阅读此书，只要勤动脑，加强理解，适当动手练习，将能够在不长的时间里掌握正则表达式的思想和技术精华，这一点已经被很多人验证过，我本人也是这本书的受益者之一。正因为这本书独一无二的地位和高度的可读性，也因为正则表达式作为一项了不起的技术发明所具有的巨大威力，我非常希望更多的读者能够通过认真地学习本书而掌握这一强大技术，并享受这项技术带来的快乐。

孟岩

2007 年 7 月于北京

译者序

《精通正则表达式（第3版）》（即 Mastering Regular Expression，3rd Edition）是一本好书。

我还记得，自己刚开始工作时，就遇到了关于正则表达式的问题（从此被逼上梁山）：若从文本中抽取E-mail地址，还可以用字符串来查找（先定位到@，然后向两端查找），若要抽取URL，简单的文本查找就无能为力了。正当我一筹莫展之时，项目经理说："可以用正则表达式，去网上找找资料吧。"抱着这根救命稻草，我搜索了之前只是听说过名字的正则表达式的资料，并打印了 `java.util.regex`（开发用的Java）的文档来看。摸索了半天，我的感觉就是，这玩意儿，真神奇，真复杂，真好用。

此后，用到正则表达式的地方越来越多，我也越来越感觉到它的重要，然而使用起来却总感觉捉襟见肘。当时是夏天，北京非常热，我决定下班之后不再着急赶车回家，而是在公司安心看看技术文档，于是邂逅了这本 *Mastering Regular Expression*。该书原文是相当通畅易懂的，看完全书大概花了我一周的业余时间，之后便如拨云见日，感觉豁然开朗——原来正则表达式可以这样用，真是奇妙，令人拍案叫绝。

此后我运用正则表达式便不用再看什么资料了，充其量就是查查语言的具体文档，表达式的基本模型和思路，完全是在阅读本书时确立的。也正是因为细心阅读过本书，所以有时我能以正则表达式解决某些复杂的问题。我的朋友郝培强（Tinyfool，昵称Tiny）曾问过我这样一个正则表达式的问题：在Apache服务器的Rewrite规则中，怎样以一个正则表达式匹配"除两个特定子域名之外的所有其他子域名"，其他人的办法都无法满足要求：要么只能匹配这两个特定的子域名，要么必须依赖程序分支才能进行判断。其实这个问题，是可以用一个正则表达式匹配的。事后，Tiny 说，看来，会用正则的人很多，但真正懂得正则的人很少。现实情况也确实如此，就我所见，不少同仁对正则表达式的运用，大多是从网上找些现成的表达式，套用在自己的程序中，但对到底该用几个反斜线转义，转义是在字符串级别还是表达式级别进行的，捕获型括号是否必须，表达式的效率如何，等等问题，往往都是一知半解，甚至毫无概念，在Tiny的问题面前，更是束手无策，一筹莫展。

就我个人来说，我所掌握的正则表达式的知识，绝大多数来自本书。正是依靠这些知识，我几乎能以正则表达式进行自己期望的任何文本处理，所以我相信，能够耐心读完这本书的读者，一定能深入正则表达式的世界，若再加以练习和思考，就能熟练地依靠它解决各种复杂的问题（其中就包括类似Tiny的问题）了。

去年，通过霍炬（Virushuo）的介绍，我参加了博文视点的试译活动，很幸运地获得了翻译本书的机会。有机会与大家分享这样一本好书，我深感荣幸。500 多页的书，拖拖拉拉，也花了半年多的时间。虽然之前读过原著，积累了一些运用正则表达式的经验，也翻译过数十万字的资料，但要尽可能准确、贴切地传达原文的阅读感觉，我仍感颇费心力。部分译文在确认理解原文的基础上，要以符合中文习惯的方式加以表述仍然颇费周折（例如，直译的"正则表达式确实容许出现这种错误"，原文的意思是"这样的错误超出了正则表达式的能力"，最后修改为"出现这样的错误，不能怪正则表达式"或"这样的问题，错不在正则表达式"）。另有部分词语，虽可译为中文，但为保证阅读的流畅，没有翻译（例如，"它包含特殊和一般两个部分，特殊部分之所以是特殊的，原因在于……"，此处 special 和 normal 是专指，故翻译为"它包含 special 和 normal 两个部分，special 部分之所以得名，原因在于……"），这样的处理，相信不会影响读者的理解。

在本书翻译结束之际，我首先要感谢霍炬，他的引荐让我获得了翻译这本书的机会；还要感谢博文视点的周筠老师，她谨慎严格的工作态度，时刻提醒我不能马虎对待这本经典之作；还有本书的责编晓菲，她为本书的编辑和校对做了大量细致而深入的工作。

另外我还要感谢东北师范大学文学院的王确老师，在我求学期间，王老师给予我诸多指点，离校时间愈长，愈是怀念和庆幸那段经历，可以说，没有与他的相识，便没有我的今天。

翻译过程中，我虽力求把握原文，语言通畅，但翻译中的错误或许是在所难免的，对此本人愿负全部责任。希望广大读者发现错误能及时与我和出版社联系以便重印时修正，或是以勘误的形式公布出来以惠及其他读者。如果读者有任何想法或建议，欢迎给我写信，我的邮件地址是：*yusheng.regex@gmail.com*。

如今正则表达式已经成为几乎所有主流编程语言中的必备元素：Java、Perl、Python、PHP、Ruby……莫不如此，甚至功能稍强大一些的文本编辑工具，都支持正则表达式。尤其是在 Web 兴起之后，开发任务中的一大部分甚至全部，都是对字符串的处理。相比简单的字符串比较、查找、替换，正则表达式提供了强大得多的处理能力（最重要的是，它能够处理"符合某种抽象模式"的字符串，而不是固化的、具体的字符串）。熟练运用它们，能够节省大量的开发时间，甚至解决一些之前看来是 mission impossible 的问题。

本书是讲解正则表达式的经典之作。其他介绍正则表达式的资料，往往局限于具体的语法和函数的讲解，于语法细节处着墨太多，忽略了正则表达式本身。这样，读者虽然对关于正则表达式的具体规定有所了解，但终究是只见树木不见森林，遇上复杂的情况，往往束手无策，举步维艰。而本书自第 1 版开始便着力于教会读者"以正则表达式来思考（think regular expression）"，向读者讲授正则表达式的精髓（正则表达式的各种流派、匹配原理、优化原则，等等），而不拘泥于具体的规定和形式。了解这些精髓，再辅以具体操作的文档，

读者便可做到“胸中有丘壑，下笔如有神”；即便问题无法以正则表达式来解决，读者也能很快作出判断，而不必盲目尝试，徒费工夫。

不了解正则表达式的读者，可循序渐进，依次阅读各章，即便之前完全未接触过正则表达式，读过前两章，也能在心中描绘出概略的图谱。第3、4、5、6章是本书的重点，也是核心价值所在，它们分别介绍了正则表达式的特性和流派、匹配原理、实用诀窍以及调校措施。这样的知识与具体语言无关，适用于几乎所有的语言和工具（当然，如果使用DFA引擎，第6章的价值要打些折扣），所谓“大象无形”，便是如此。读者如能仔细研读，悉心揣摩，之后解决各种问题时，必定获益匪浅。第7、8、9、10章分别讲解了Perl、Java、.NET、PHP中正则表达式的用法，看来类似参考手册，其实是对前面4章知识的包装，将抽象的知识辅以具体的语言规定，以具体的形式表现出来。所以，心急的读者，在阅读这些章节之前，最好先通读第3、4、5、6章，以便更好地理解其中的逻辑和思路。

相信仔细阅读完本书的读者，定会有登堂入室的感觉。不但能见识到正则表达式各种令人眼花缭乱的特性，更能够深入了解表达式、匹配、引擎背后的原理，从而写出复杂、神奇而又高效的正则表达式，快速地解决工作中的各种问题。

余晟

2007年6月于北京



前言

Preface

本书关注的是一种强大的工具——“正则表达式”。它将教会读者如何使用正则表达式解决各种问题，以及如何充分使用支持正则表达式的工具和语言。许多关于正则表达式的文档都没有介绍这种工具的能力，而本书的目的正是让读者“**精通**”正则表达式。

许多种工具都支持正则表达式（文本编辑器、文字处理软件、系统工具、数据库引擎，等等），不过，要想充分挖掘正则表达式的能力，还是应当将它作为编程语言的一部分。例如Java、JScript、Visual Basic、VBScript、JavaScript、ECMAScript、C、C++、C#、elisp、Perl、Python、Tcl、Ruby、PHP、*sed*和*awk*。事实上，在一些用上述语言编写的程序中，正则表达式扮演了极其重要的角色。

正则表达式能够得到众多语言和工具的支持是有原因的：它们极其有用。从较低的层面上来说，正则表达式描述的是一串文本（a chunk of text）的特征。读者可以用它来验证用户输入的数据，或者也可以用它来检索大量的文本。从较高的层面上来说，正则表达式容许用户掌控他们自己的数据——控制这些数据，让它们为自己服务。掌握正则表达式，就是掌握自己的数据。

本书的价值

The Need for This Book

本书的第1版写于1996年，以满足当时存在的需求。那时还没有关于正则表达式的详尽文档，所以它的大部分能力还没有被发掘出来。正则表达式文档倒是存在，但它们都立足于“低层次视角”。我认为，那种情况就好像是教一些人英文字母，然后就指望他们会说话。

第 2 版与第 1 版间隔了五年半的时间，这期间，互联网迅速流行起来，正则表达式的形式也有了极大的扩张，这或许并不是巧合。几乎所有工具软件和程序语言支持的正则表达式也变得更加强大和易于使用。Perl、Python、Tcl、Java 和 Visual Basic 都提供了新的正则支持。新出现的支持内建正则表达式的语言，例如 PHP、Ruby、C#，也已经发展壮大，流行开来。在这段时间里，本书的核心——如何真正理解正则表达式，以及如何使用正则表达式——仍然保持着它的重要性和参考价值。

不过，第 1 版已经逐渐脱离了时代，必须加以修订，才能适应新的语言和特性，也才能对应正则表达式在互联网世界中越来越重要的地位。第 2 版出版于 2002 年，这一年的里程碑是 `java.util.regex`、Microsoft .NET Framework 和 Perl 5.8 的诞生。第 2 版全面覆盖了这些内容。关于第 2 版，我唯一的遗憾就是，它没有提及 PHP。自第 2 版诞生以来的 4 年里，PHP 的重要性一直在增加，所以，弥补这一缺憾是非常迫切的。

第 3 版在前面的章节中增加了 PHP 的相关内容，并专门为理解和应用 PHP 的正则表达式增加了一章全新的内容。另外，该版对 Java 的章节也进行了修订，做了可观的扩充，反映了 Java1.5 和 Java1.6 的新特性。

目标读者

Intended Audience

任何有机会使用正则表达式的人，都会对本书感兴趣。如果您还不了解正则表达式能提供的强大功能，这本书展示的全新世界将会让您受益匪浅。即使您认为自己已经是掌握正则表达式的高手了，这本书也能够深化您的认识。第 1 版面世后，我时常会收到读者的电子邮件反映说“读这本书之前，我**以为**自己了解正则表达式，但**现在**我才真正了解”。

以与文本打交道为工作（如 Web 开发）的程序员将会发现，这本书绝对称得上是座金矿，因为其中蕴藏了各种细节、暗示、讲解，以及能够立刻投入到实用中的知识。在其他任何地方都难以找到这样丰富的细节。

正则表达式是一种思想——各种工具以各种方式（数目远远超过本书的列举）来实现它。如果读者理解了正则表达式的基本思想，掌握某种特殊的实现就是易如反掌的事情。本书关注的就是这种思想，所以其中的许多知识并不受例子中所用的工具软件和语言的束缚。

如何阅读

How to Read This Book

这本书既是教程，又是参考手册，还可以当故事看，这取决于读者的阅读方式。熟悉正则表达式的读者可能会觉得，这本书马上就能当作一本详细的参考手册，读者可以直接跳到自己需要的章节。不过，我并不鼓励这样做。

要想充分利用这本书，可以把前 6 章作为故事来读。我发现，某些思维习惯和思维方式的确有助于完整的理解，不过最好还是从这几章的讲解中学习它们，而不是仅仅记住其中的几张列表。

故事是这样的，前 6 章是后面 4 章——包括 Perl、Java、.NET 和 PHP——的基础。为了帮助读者理解每一部分，我交叉使用各章的知识，为了提供尽可能方便的索引，我投入了大量的精力（全书中有超过 1 200 处交叉引用，它们以符号加页码的形式标注）。

在读完整个故事以前，最好不要把本书作为参考手册。在开始阅读之前，读者可以参考其中的表格，例如第 92 页的图表，想象它代表了需要掌握的相关信息。但是，还有大量背景信息没有包含在图表中，而是隐藏在故事里。读者阅读完整个故事之后，会对这些问题有个清晰的概念，哪些能够记起来，哪些需要温习。

组织结构

Organization

全书共 10 章，可以从逻辑上粗略地分为三类，下面是总体概览：

导引

第 1 章：介绍正则表达式的基本概念。

第 2 章：考察利用正则表达式进行文本处理的过程。

第 3 章：提供对于特性和工具软件的概述以及简史。

细节

第 4 章：揭示了正则表达式的工作原理的细节。

第 5 章：利用第 4 章的知识，继续学习各种例子。

第 6 章：详细讨论效率问题。

特定工具的知识

第 7 章：详细讲解 Perl 的正则表达式。

第 8 章：讲解 Sun 提供的 `java.util.regex` 包。

第 9 章：讲解.NET 的语言中立的正则表达式包。

第 10 章：讲解 PHP 中提供正则功能的 preg 套件。



导引部分会把完全的门外汉变成“对问题有感觉”的新手。对正则表达式有一定经验的读者完全可以快速翻阅这些章节，不过，即使是对于相当有经验的读者来说，我仍然要特别推荐第 3 章。

- **第 1 章 正则表达式入门**，是为完全的门外汉准备的。我以应用相当广泛的程序 *egrep* 为例来介绍正则表达式，我也提供了我的视角：如何“理解”正则表达式，来为后面章节所包括的高级概念打下坚实的基础。即使是有经验的读者，浏览本章也会有所收获。

- **第 2 章 入门示例拓展**，考察了支持正则表达式的程序设计语言的真实文本处理过程。附加的示例提供了后面章节详细讨论的基础，也展示了高级正则表达式调校背后的重要思考过程。为了让读者学会“正则表达式的套路”，这章出现了一个复杂问题，并讲解了两种全然不相关的工具如何分别通过正则表达式来解决它。

- **第 3 章 正则表达式的特性和流派概览**，提供了当前经常使用的工具的多种正则表达式的概览。因为历史的混乱，当前常用的正则表达式的类型可能差异巨大。此章同时介绍了正则表达式以及使用正则表达式的工具的历史和演化历程。本章末尾也提供了“高级话题引导”。此引导是读者学习此后高级内容的路线图。

细节

The Details

了解了基础知识之后，读者需要弄明白“如何使用”及“这么做的原因”。就像“授人以渔”的典故一样，真正懂得正则表达式的读者，能够在任何时间、任何地点应用关于它的知识。

- **第 4 章 表达式的匹配原理**，循序渐进地导入本书的核心。它从实践的角度出发，考察了正则引擎真实工作的重要的内在机制。懂得正则表达式如何处理工作细节，对读者掌握它们大有裨益。

- **第 5 章 正则表达式实用技巧**，教育读者在高层次和实际的运用中应用知识。这一章会详细讲解常见（但复杂）的问题，目的在于拓展和深化读者对于正则表达式的认识。

- **第 6 章 打造高效正则表达式**，考察真实生活中大多数程序设计语言提供的正则表达式的高效结果。本章运用第 4 章和第 5 章详细讲解的知识，来开发引擎的能力，并避免其中的缺陷。

特定工具的知识

Tool-Specific Information

学习完第 4、5、6 章的读者，不太需要知道特定的实现。不过，我还是用了 4 个整章来讲解 4 种流行的语言。

- **第 7 章 Perl**，详细讲解了 Perl 的正则表达式，Perl 大概是目前最流行的主要的正则表达式编程语言。在 Perl 中，与正则表达式相关的操作符只有四个，但它们组合出的选项和特殊情形带来了大量的程序选项——同时还有陷阱。对没有经验的开发人员来说，这种极其丰富的选项能够让他们迅速从概念转向程序，当然也可能是雷场。本章的详细介绍有助于给读者指出一条光明大道。
- **第 8 章 Java**，详细介绍了 `java.util.regex` 包，从 Java 1.4 以后，它已经成为了 Java 语言的标准部分。本章主要关注的是 Java 1.5，但也提及了它与 Java 1.4.2 和 Java 1.6 的差别。
- **第 9 章 .NET**，是微软尚未提供的.NET 正则表达式库的文档。无论使用 VB.NET、C#、C++、JScript、VBScript、ECMAScript 还是使用.NET 组件的其他语言，本章都提供了详细内容，让读者能够充分利用.NET 的正则表达式。
- **第 10 章 PHP**，简要介绍了 PHP 内嵌的多个正则引擎，并详细介绍了 preg 正则表达式套件（regex engine）的类型和 API，这些是由 PCRE 正则表达式库提供的。

体例说明

Typographical Conventions

在进行（或者谈论）详细的和复杂的文本处理时，保持精确性是很重要的。差一个空格字符，可能导致截然不同的结果，所以我会在本书中使用下面的惯例：

- 正则表达式以「`this`」的形式出现。两端的符号表示“里面有一个正则表达式”，而正则表达式文字（例如用来检索的表达式）以‘`this`’的形式出现。有时候，省略两端的符号和单引号也不会造成歧义，此时我会省略它们。同样，屏幕截图通常以原来的样子出现，而不会用到上面两种符号。

- 在文字文本和表达式内部的省略号会被特别标出。例如，[···]表示一对方括号，之间的内容无关紧要，而[...]表示一对方括号，其中包含三个句点。
- 如果没有明确数字，可能很难判断“a   b”之间有多少空格，所以出现在正则表达式和文字文本中的空格以“·”表示。这样“a···b”就清楚多了。
- 我使用可见的制表符，换行符和回车字符：

·	空格字符
␉	制表符
␊	换行符
␍	回车字符

- 有时候，我会使用下画线或有色背景高亮标注文字文本或正则表达式的一部分。下面这句话中，下画线的部分表示表达式真正匹配的部分：

  Because 「cat」 matches ‘It·indicates·your·cat·is···’ instead of the word ‘cat’, we realize...

  这个例子中，下画线的部分高亮标记了表达式中添加的字符：

  To make this useful, we can wrap 「Subject|Date」 with parentheses, and append a colon and a space. This yields 「(Subject|Date):·」

- 本书包含了大量的细节和例子，所以我设置了超过 1 200 处的交叉引用，帮助读者理解。它们通常表示为“☞123”，意思是“请参阅第 123 页”。举个例子：“…的说明在表 8-2 中（☞367）”。

练习

Exercises

有时候我会问个问题，帮助读者理解正在讲解的概念，尤其是在前几章这种问题更多。它们并不是摆设，我希望读者在继续阅读之前认真想想。请记住我的话，不要忽略它们的重要意义，本书中这样的问题并不多。它们可以当作自我测试题：如果不是几句话就能说明白的问题，最好是在复习相关章节之后再继续阅读。

为了避免读者直接看到问题的答案，我使用了一点技巧：问题的答案都必须翻页才能看到。通常你必须翻过一页才能看到标着❖的答案。这样答案在你思考问题的时候没法直接看到，但又很容易获得。

链接、代码、勘误及联系方式

Links, Code, Errata, and Contacts

写第 1 版时，我发现修改书本上的 URL，保持与实际一致是件很费工夫的事情，所以，我没有在书后罗列一个 URL 附录，而是提供统一的地址：

http://regex.info

在这里你可以找到与正则表达式相关的链接，书中的所有代码，可检索的索引以及其他资源。本书也可能存在错误☺，所以我提供了勘误。

如果你找到书中的错误，或者仅仅是希望给我写几句话，请写邮件到：*jfriedl@regex.info*。

我们已尽力核验本书所提供的信息，尽管如此，仍不能保证本书完全没有瑕疵，而网络世界的变化之快，也使得本书永不过时的保证成为不可能。如果读者发现本书内容上的错误，不管是赘字、错字、语意不清，甚至是技术错误，我们都竭诚虚心接受读者指教。如果您有任何问题，请按照以下的联系方式与我们联系。

奥莱理软件（北京）有限公司

北京市 海淀区 知春路 49 号 希格玛公寓 B 座 809 室
邮政编码：100080
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

与本书有关的在线信息如下所示。

http://www.oreilly.com/catalog/regex3/（原书）
http://www.oreilly.com.cn/book.php?bn=978-7-121-04684-1（中文版）

北京博文视点资讯有限公司（武汉分部）

湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室
邮政编码：430074
电话：（027）87690813 传真：（027）87690813 转 817
网页：*http://bv.csdn.net*
读者服务信箱：
sheguang@broadview.com.cn（读者信箱）
broadvieweditor@gmail.com（投稿信箱）

目录

Table of Contents



计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 **|** 软考历年真是解析与答案

软考视频 **|** 考试机构 **|** 考试时间安排

Java 一览无余：**Java** 视频教程 **| Java SE | Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书**+**视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料**(**电子书**+**视频教程**)**下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 **| SQL Server** 篇 **| Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具**+**电子书**+**视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集**(**电子书**+**视频教程**) Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料**<**电子书**+**视频**>**分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍**+**视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

第1章 正则表达式入门

Introduction to Regular Expressions

想象一下这幅图景：你需要检索某台 Web 服务器上的页面中的重复单词（例如“this this”），进行大规模文本编辑时，这是一项常见的任务。程序必须满足下面的要求：

- 能检查多个文件，挑出包含重复单词的行，高亮标记每个重复单词（使用标准 ANSI 的转义字符序列（escape sequence）），同时必须显示这行文字来自哪个文件。
- 能跨行查找，即使两个单词一个在某行末尾而另一个在下一行的开头，也算重复单词。
- 能进行不区分大小写的查找，例如‘`The the…`’，重复单词之间可以出现任意数量的**空白字符**（空格符、制表符、换行符之类）（译注1）。
- 能查找用 HTML tag 分隔的重复单词。HTML tag 用于标记互联网页上的文本，例如，粗体单词是这样表示的：‘`…it is <B>very</B> very important…`’。

这些问题并不容易解决，但又不能不解决。我在写作本书的手稿时，曾用一个工具来检查已经写好的部分，我惊奇地发现，其中竟有那么多的重复单词。能够解决这种问题的编程语言有许多，但是用支持正则表达式的语言来处理会相当简单。

正则表达式（Regular Expression）是强大、便捷、高效的文本处理工具。正则表达式本身，加上如同一门袖珍编程语言的通用模式表示法（general pattern notation），赋予使用者描述和分析文本的能力。配合上特定工具提供的额外支持，正则表达式能够添加、删除、分离、叠加、插入和修整各种类型的文本和数据。

译注1：whitespace 泛指“空白”，space 特指“空格”，在下文中，将两者分别翻译为“空白字符”与“空格符”。

正则表达式的使用难度只相当于文本编辑器的搜索命令，但功能却与完整的文本处理语言一样强大。本书将向读者展示正则表达式提高生产率的诸多办法。它会教导读者如何学会用正则表达式来**思考**（think regular expressions），以便于掌握它们，充分利用它们的强大功能。

如果使用当今流行的程序设计语言，解决重复单词问题的完整程序可能仅仅只需要几行代码。使用一个正则表达式的搜索和替换命令，读者就可以查找文档中的重复单词，并把它们标记为高亮。加上另一个，你可以删除所有不包含重复单词的行（只留下需要在结果中出现的行）。最后，利用第三个正则表达式，你可以确保结果中的所有行都以它所在文件的名字开头。在下一章里，我们会看到用 Perl 和 Java 编写的程序。

宿主语言（例如 Perl、Java 以及 VB.NET）提供了外围的处理支持，但是真正的能力来自正则表达式。为了驾驭这种语言，满足自己的需求，读者必须知道如何构建正则表达式，才能识别符合要求的文本，同时忽略不需要的文本。然后，就可以把表达式和语言支持的构建方式结合起来，真正处理这些文本（加入合适的高亮标记代码，删除文本，修改文本，等等）。

解决实际问题

Solving Real Problems

掌握正则表达式，可能带来超乎你之前想象的文本处理能力。每一天，我都依靠正则表达式解决各种大大小小的问题（通常的情况是，问题本身并不复杂，但没有正则表达式就成了大问题）。

要说明正则表达式的价值，可以举一个用正则表达式解决大而重要的问题的例子，但是它不一定能代表正则表达式在平时解决的那些“不值一提”（uninteresting）的问题。这里的“不值一提”是指这类问题并不能成为谈资，可是不解决它们，你就没法继续干活。

举个简单的例子，我需要检查许多文件（事实上，本书的手稿存放在 70 个文件中），确保每一行中 ‘`SetSize`’ 出现的次数与 ‘`ResetSize`’ 的一样多。为了应付复杂的情况，我还需要考虑大小写的情况（举例来说，‘`setSIZE`’ 也算做 ‘`SetSize`’）。人工检查 32 000 行文字显然不现实。

即便使用文本编辑器的“单词查找”功能，也不够方便，尤其是对所有文件进行同样的操作，何况还需要考虑所有可能的大小写情况。

正则表达式就是解决这个问题的灵丹妙药。只需要一个简单的命令，我就能够检查所有的文件，获得我需要知道的结果。时间是：写命令大概 15 秒，检索所有的数据实际只花了 2 秒。这真是棒极了（如果您想知道这是怎么做到的，不妨现在就翻到第 36 页）！

再举一个例子，我曾帮助一个朋友处理远端机器上的某些 E-mail，他希望我把他邮箱文件中的消息作为列表发送给他。我可以把整个文件导入文本编辑器，手工删除所有信息，只留下邮件头中的几行，作为内容的列表。尽管文件不是很大，连接速度也不算慢，这样的任务还是很耗费时间而且很乏味。而且，窥见他的邮件正文，也令我尴尬。

正则表达式再一次提供了帮助！我用一个简单的命令（使用本章稍后提到的一个常用工具 *egrep*）显示每封邮件的 `From:`和 `Subject:`字段。为了告诉 *egrep* 我需要提取哪些行，我使用了正则表达式「`^(From|Sbuject):`」。

朋友得到这个列表之后，让我找一封特殊的（5 000 行！）邮件。使用文本编辑器或者邮件系统来提取一封邮件无疑非常耗时。相反，我借助另一个工具（叫做 *sed*），同样使用正则表达式来描述文件中我需要的内容。这样，我能迅速而方便地提取和发送需要的邮件。

使用正则表达式节省下来的时间或许并不能让人“激动”，但总比把时间消耗在文本编辑器中要好。如果我不知道有正则表达式这种玩意儿，根本就不会想到还有别的解决办法。所以，这个故事告诉我们，正则表达式和相关的工具能够让我们以可能未曾想过的方式来解决问题。

一旦掌握了正则表达式，你就会知道到它简直是工具中的无价之宝，你也难以想象之前那些没有正则表达式的日子是怎么度过的（注 1）。

全面掌握正则表达式是很有用的。本书提供了掌握这种技能所需要的信息，我同时也希望，这本书也提供了促使你学习的动机。

注 1：用过 TiVo（译注：TiVo 是一种数字录像机，具有许多神奇的功能，例如根据用户的偏好自动录制节目，自动跳过电视台的广告，等等）的人都体验过这种感觉。

作为编程语言的正则表达式

Regular Expressions as a Language

如果没有正则表达式相关经验，读者可能无法理解上个例子中正则表达式「^(From|Subject):」的意义，但是这个表达式并没有什么神奇之处。其实魔术本身也不神奇，只是缺乏训练的普通观众不明白魔术师掌握的那些技巧而已。如果你也懂得如何在手中藏一张牌，那么，熟练之后，你也可以"变魔术"。外语也是这样——一旦掌握了一门外语，你就不会觉得它像天书了。

以文件名做类比

The Filename Analogy

选择这本书的读者，大概对"正则表达式" 多少有点认识。即便没有，也应该熟悉其中的基本概念。

我们都知道，*report.txt* 是一个文件名，但是，如果你用过 Unix 或者 DOS/Windows 的话，就会知道"*.txt"能够用来选择多个文件。在此类文件名（称为**"文件群组"** file globs 或者**"通配符"** wildcards）中，有些字符具有特殊的意义。星号表示"任意文本"，问号表示"任意单个字符"。所以，文件群组"*.txt"以能够匹配字符的「*」符号开头，以普通文字「.txt」结尾，所以，它的意思是：选择以任意文本开头，以.txt 结尾的所有文件。

大多数系统都提供了少量的附加特殊字符（additional special characters），但是，总的来说，这些文件名模式（filename patterns）的表达能力还很有限。不过，因为这类问题的领域很狭窄——只涉及文件名，所以这算不上缺陷。

不过，处理普通的文本就没有这么简单了。散文、诗、程序代码、报表、HTML、表格、单词表……到你想得出的任何文本。如果某种特殊的需求足够专业，例如"选择文件"，我们可以发明一些特殊的办法和工具来解决问题。不过，近年来，一种**"通用的模式语言"**（generalized pattern language）已经发展起来，它功能强大，描述能力也很强，可以用来解决各种问题。不同的程序以不同的方式来实现和使用这种语言，但是综合来说，这种功能强大的模式语言和模式本身被称为**"正则表达式"**（regular expression）。

以语言做类比

The Language Analogy

完整的正则表达式由两种字符构成。特殊字符（special characters，例如文件名例子中的*）称为“**元字符**”（metacharacters），其他为“**文字**”（literal），或者是普通文本字符（normal text characters）。正则表达式与文件名模式（filename pattern）的区别就在于，正则表达式的元字符提供了更强大的描述能力。文件名模式只为有限的需求提供了有限的元字符，但是正则表达式“语言”为高级应用提供了丰富而且描述力极强的元字符。

为了便于理解，我们可以把正则表达式想象为普通的语言，普通字符对应普通语言中的单词，而元字符对应语法。根据语言的规则，按照语法把单词组合起来，就会得到能传达思想的文本。在 E-mail 的例子中，我用正则表达式「`^(From|Subject):`」来寻找以‘`From:`’或者‘`Subject:`’开头的行。下画线标注的就是特殊字符，稍后我们将解释它们的含义。

就像学习任何一门外语一样，第一眼看上去，正则表达式很不好理解。这也是那些对它只有粗浅了解或者根本不了解的人觉得正则表达式很神奇的原因。但是，就像学日语的人很快就能理解正規表現は簡単だよ！（注 2）一样，读者很快也能够彻底明白下面这个正则表达式的含义：

```
s!<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>!<inet>$1</inet>!
```

这个例子取自一个 Perl 脚本，我的编辑器用它来修改手稿。手稿的作者错误地使用了`<emphasis>`这个 tag 来标注 IP 地址（类似`209.204.146.22`这样由数字和点号构成的字符串）。其中的奥妙就在于使用 Perl 的文本替换命令，使用：

```
「<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>」
```

把 IP 地址两端的 tag 替换为`<inet>`，而不改动其他的`<emphasis>`标签。在后面的章节中，读者会了解这个表达式的构造细节，然后就能按照自己的需求，在自己的应用程序或者开发语言中应用这些技巧。

注 2：这句话的意思是，“正则表达式很简单！”。有趣的是，就像第 3 章介绍的，“正则表达式”这个术语来自形式代数。问我这本书的主题的人，如果对这个概念不熟悉，听到“正则表达式”多半会满脸茫然。正则表达式在日文中写作，正規表現，同它的英文名字一样不好理解，但是我用日语来回答通常会令人反应更奇怪。因为在日文中，“正则”（regular）很不幸地与一个表示“生殖器官”的医学术语发音相同。读者可以想象，在我没有解释之前，人们会有多么惊奇。

本书的目的

你或许不需要重复把`<emphasis>`替换为`<inet>`的工作，不过很可能需要解决“把**这些**文字替换为**那些**文字”的问题。本书的目的不是提供具体问题的解决办法，而是教会读者利用正则表达式来**思考**，解决遇到的各种问题。

正则表达式的思维框架

The Regular-Expression Frame of Mind

我们将会看到，完整的正则表达式由小的构建模块单元（building block unit）组成。每个单独的构建模块都很简单，不过因为它们能够以无穷多种方式组合，将它们结合起来实现特殊目标必须依靠经验。所以，本章提供了有关正则表达式的若干概念的总体描述。这一章并没有艰深的内容，而是为本书其余章节的知识打下基础，在深入探索正则表达式之前，把相关事宜阐释清楚。

某些例子看起来可能有点无聊（因为它们确实无聊），但它们代表了一类需要完成的任务，只是读者目前可能还没有意识到。即使觉得每个例子的意义都不大也不必担心，慢慢理解其中的道理就好。这就是本章的目的。

对于有部分经验的读者

If You Have Some Regular-Expression Experience

如果读者已经熟悉正则表达式，这些综述便没有太大价值，但务必不要忽略它们。你或许明白某些元字符的基本意义，但某些思维和看待正则表达式的方式可能是你不了解的。

就像真正**懂演奏**和仅仅会弹奏之间差别迥异一样，了解正则表达式和**真正理解**正则表达式并不是一回事。某些内容可能会重复读者已经了解的知识，但方式可能与之前的不同，而且这些方式正是**真正理解**正则表达式的第一步。

检索文本文件：Egrep

Searching Text Files: Egrep

文本检索是正则表达式最简单的应用之一——许多文本编辑器和文字处理软件都提供了正则表达式检索的功能。最简单的就是 *egrep*。在指定了正则表达式和需要检索的文件之后，*egrep* 会尝试用正则表达式来匹配每个文件的每一行，并显示能够匹配的行。

许多系统——例如 DOS、MacOS、Windows、Unix 等等——都对应有免费提供的 *egrep*。在本书的网页 *http://regex.info* 上可以找到获得对应读者操作系统的 *egrep* 拷贝的链接。

回到第 3 页的 E-mail 的例子，真正用来从 E-mail 文件中提取结果的命令如图 1-1 所示。*egrep* 把第一个命令行参数视为一个正则表达式，剩下的参数作为待搜检索的文件名。注意，图 1-1 中的单引号**并不是**正则表达式的一部分，而是根据 command shell 需要添加的（注 3）。使用 *egrep* 时，我通常用单引号来包围正则表达式。如果要在支持对正则表达式提供了完整支持的程序设计语言中使用正则表达式——这是下一章开头的内容，重要的问题是知道特殊字符有哪些，具体文本是什么，针对什么对象（什么表达式，什么工具软件），以及按何种顺序解释这些字符。

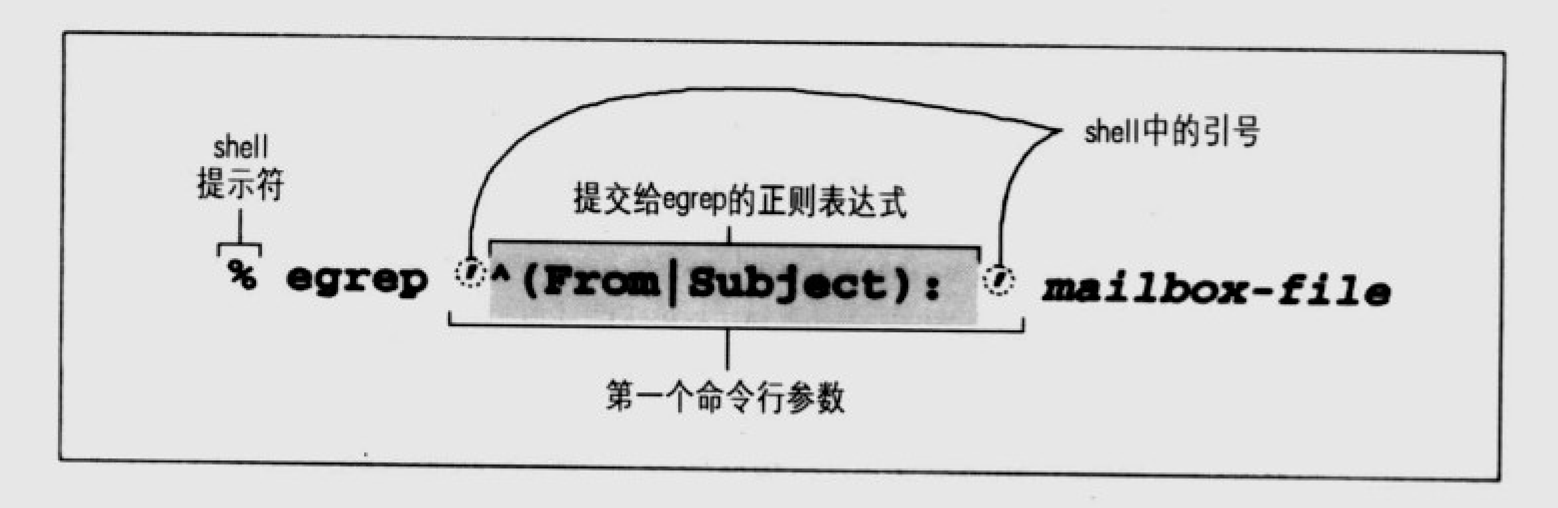


图 1-1：通过命令行调用 egrep

我们马上就能明白，这个正则表达式的各个部分都是什么意思，但已经知道某些字符具有特殊含义的读者或许能够猜出大概了。在这里，「^」和「|」都是正则表达式的元字符，它们与其他字符结合起来，实现我们期望的功能。

如果一个正则表达式不包括任何 *egrep* 支持的元字符，它就成了一个简单的“纯文本”检索。例如，在一个文件中检索「cat」，会显示任何包含 c·a·t 这 3 个连续字母的行。例如，它包括所有出现了 vacation 的行。

注 3：command shell 是操作系统的一部分，用来接收用户的命令，执行用户请求的程序。在我使用的 shell 中，单引号用来分组命令参数，告诉 shell 不必关心其中的内容。如果不这样写，shell 或许会认为，我在正则表达式中使用的‘*’是需要解释的文件名模式的一部分。而这不是我的意思，所以我用单引号在 shell 中“屏蔽（hide）”元字符。使用 COMMAND.COM 或者 CMD.EXE 的 Windows 用户可能需要使用双引号而不是单引号。

即便这行文本中不包含**单词** cat，vacation 中包含的 c·a·t 序列仍然符合匹配条件。如果某行中包含 vacation，*egrep* 就会把它显示出来。关键就在于，此处进行的正则表达式搜索不是基于"单词"的——*egrep* 能够理解文件中的字节和行，但它完全不理解英语（或者其他任何语言）的单词、句子、段落，或者是其他复杂概念。

Egrep 元字符

Egrep Metacharacters

现在我们来看 *egrep* 中支持正则表达式功能的元字符。我会用几个例子来简要介绍它们，把详细的例子和描述留到后面的章节。

印刷体例 在开始之前，请务必回顾前言第 V 页上解释的体例说明。本书使用了一些新的文字形式，所以某些体例读者初次接触可能并不熟悉。

行的起始和结束

Start and End of the Line

或许最容易理解的元字符就是**脱字符号**「^」和**美元符号**「$」了，在检查一行文本时，「^」代表一行的开始，「$」代表结束。我们曾经看到，正则表达式「cat」寻找的是一行文本中任意位置的 c·a·t，但是「^cat」只寻找行首的 c·a·t——「^」用来把匹配文本（这个表达式的其他部分匹配的字符）**"锚定"**（anchor）在这一行的开头。同样，「cat$」只寻找位于行末的 c·a·t，例如以 scat 结尾的行。

读者最好能养成按照字符来理解正则表达式的习惯。例如，不要这样：

> 「^cat」匹配以 cat 开头的行

而应该这样理解：

> 「^cat」匹配的是以 c 作为一行的第一个字符，紧接一个 a，紧接一个 t 的文本。

这两种理解的结果并无差异，但按照字符来解读更易于明白新遇到的正则表达式的内部逻辑。*egrep* 会如何解释「^cat$」、「^$」和单个的「^」呢？◆ 请翻到下页查看答案。

脱字符号和美元符号的特别之处就在于，它们匹配的是一个**位置**，而不是具体的文本。当然，有很多方式可以匹配具体文本。在正则表达式中，除了使用「cat」之类的普通字符，还可以使用下面几节介绍的元字符。

字符组

Character Classes

匹配若干字符之一

如果我们需要搜索的是单词“grey”，同时又不确定它是否写作“gray”，就可以使用正则表达式结构体（construct）「[…]」。它容许使用者列出在某处期望匹配的字符，通常被称作**字符组**（character class（译注 2））。「e」匹配字符 e，「a」匹配字符 a，而正则表达式「[ea]」能匹配 a 或者 e。所以，「gr[ea]y」的意思是：先找到 g，跟着是一个 r，然后是一个 a 或者 e，最后是一个 y。我很不擅长拼写，所以总是用正则表达式从一大堆英文单词中找到正确的拼写。我经常使用的一个正则表达式是「sep[ea]r[ea]te」，因为我从来都记不住这个单词到底是写作“seperate”，“separate”，“separete”，还是别的什么样子。匹配的结果的就是正确的拼法，而正则表达式就是我的领路人。

请注意，在字符组以外，普通字符（例如「gr[ae]y」中的「g」和「r」）都有“接下来是（and then）”的意思——“首先匹配「g」，**接下来是**「r」……”。这与字符组内部的情况是完全相反的。字符组的内容是在同一个位置能够匹配的若干字符，所以它的意思是“**或**”。

来看另一个例子，我们还必须考虑单词的第一个字母为大写的情况，例如「[Ss]mith」。请记住，这个表达式仍然能够匹配内嵌在其他单词里头的 smith（或者是 Smith），例如 blacksmith。在综述阶段，我不打算为这种情况费太多笔墨，但是这确实是某些新手遇到的问题的根源。等了解了更多的元字符以后，我会介绍一些办法来解决单词嵌套的问题。

在一个字符组中可以列举任意多个字符。例如「[123456]」匹配 1 到 6 中的任意一个数字。这个字符组可以作为「<H[123456]>」的一部分，用来匹配<H1>、<H2>、<H3>等等。在搜索 HTML 代码的头文件时这非常有用。

在字符组内部，**字符组元字符**（character-class metacharacter）‘-’（连字符）表示一个范围：「<H[1-6]>」与「<H[123456]>」是完全一样的。「[0-9]」和「[a-z]」是常用的匹配数字和小写字母的简便方式。多重范围也是容许的，例如「[0123456789abcdefABCDEF]」可以写作「[0-9a-fA-F]」（或者也可以写作「[A-Fa-f0-9]」，顺序无所谓）。这 3 个正则表达式非常适用于处理十六进制数字。我们还可以随心所欲地把字符范围与普通文本结合起来：「[0-9A-Z_!.?]」能够匹配一个数字、大写字母、下画线、惊叹号、点号，或者是问号。

请注意，只有在字符组内部，连字符才是元字符——否则它就只能匹配普通的连字符号。其实，即使在字符组内部，它也不一定就是元字符。如果连字符出现在字符组的开头，它表示的就只是一个普通字符，而不是一个范围。同样的道理，问号和点号通常被当作元字符处理，但在字符组中则不是如此（说明白一点就是，「[0-9A-Z_!.?]」里面，真正的特殊字符就只有那两个连字符）。

译注 2：台湾翻译为“字符集”，但通常“字符集”指的是 character set，为避免混淆，此处翻译为“字符组”。

分析「^cat$」、「^$」和「^」

◆ 第8页问题的答案

「^cat$」 **文字意义**：匹配的条件是，行开头（显然，每一行都有开头），然后是字母c·a·t，然后是行末尾。

应用意义：只包含cat的行——没有多余的单词、空白字符……只有'cat'。

「^$」 **文字意义**：匹配的条件是，行开头，然后就是行末尾。

应用意义：空行（没有任何字符，包括空白字符）。

「^」 **文字意义**：匹配条件是行的开头。

应用意义：**无意义！**因为每一行都有开头，所以每一行都能匹配——空行也不例外。

> 不妨把字符组看作独立的微型语言。在字符组内部和外部，关于元字符的规定（哪些是元字符，以及它们的意义）是不同的。

我们很快就会看到更多的例子。

排除型字符组

用「[^…]」取代「[…]」，这个字符组就会匹配任何**未**列出的字符。例如，「[^1-6]」匹配除了1到6**以外**的任何字符。这个字符组中开头的「^」表示"排除（negate）"，所以这里列出的不是希望匹配的字符，而是不希望匹配的字符。

读者可能注意到了，这里的^和第8页的表示行首的脱字符是一样的。字符确实相同，但意义截然不同。英语里的"wind"，根据情境的不同，可能表示一阵强烈的气流（风），也可能表示给钟表上发条；元字符也是如此。我们已经看过用来表示范围的连字符的例子。只有在字符组内部（而且不是第一个字符的情况下），连字符才能表示范围。在字符组外部，^表示一个行锚点（line anchor），但是在字符组内部（而且必须是紧接在字符组的第一个方括号之后），它就是一个元字符。请不要担心——这就是最复杂的情况，接下来的内容比这简单。

来看另一个例子，我们需要在一堆英文单词中搜索出一些特殊的单词：在这些单词中，字母 q 后面的字母不是 u。用正则表达式来表示，就是「q[^u]」。用这个正则表达式来搜索我手头的数据，确实得到了一些结果，但显然不多，其中还有些是我没见过的英文单词。

下面是结果（我输入的命令用粗体表示）：

```
% egrep 'q[^u]' word.list
Iraqi
Iraqian
miqra
qasida
qintar
qoph
zaqqum%
```

其中有两个单词值得注意：伊拉克“Iraq”和澳大利亚航空公司的名字“Qantas”。尽管它们都在 *word.list* 文件中，但都不包含在 *egrep* 结果中。为什么呢？◆请动动脑筋，然后翻到下一页来检查你的答案。

请记住，排除型字符组表示“匹配一个未列出的字符（match a character that's not listed）”，而不是“不要匹配列出的字符（don't match what is listed）”。这两种说法看起来一样，但是 Iraq 的例子说明了其中的细微差异。有一种简单的理解排除型字符组的办法，就是把它们看作普通的字符组，里面包含的是除了“排除型字符组中所有字符”以外的字符。

用点号匹配任意字符

Matching Any Character with Dot

元字符「.」（通常称为点号 *dot* 或者小点 *point*）是用来匹配任意字符的字符组的简便写法。如果我们需要在表达式中使用一个“匹配任何字符”的占位符（placeholder），用点号就很方便。例如，如果我们需要搜索 03/19/76、03-19-76 或者 03.19.76，不怕麻烦的话用一个明确容许‘/’、‘-’、‘.’的字符组来构建正则表达式，例如「03[-./]19[-./]76」。也可以简单地尝试「03.19.76」。

读者第一次接触这个表达式时，可能还不清楚某些情况。在「03[-./]19[-./]76」中，点号并不是元字符，因为它们在字符组内部（记住，在字符组里面和外面，元字符的定义和意义是不一样的）。这里的连字符同样也不是元字符，因为它们都紧接在 [或者 [^ 之后。如果连字符不在字符组的开头，例如「[.-/]」，就是用来表示范围的，在本例中就是错误的用法。

测验答案

◆ 第11页问题的答案

为什么「q[^u]」无法匹配'Qantas'或者'Iraq'？

Qantas无法匹配的原因是，正则表达式要求小写q，而Qantas中的Q是大写的。如果我们改用Q[^u]，就能匹配Qantas，但是其他单词又不在结果中了，因为它们不包括大写Q。[Qq][^u]则能找到上面所有的单词。

Iraq的例子有点迷惑人。正则表达式要求q**之后紧跟一个**u以外的字符，这就排除了q**处在行尾**的情况。通常来说，文本行的结尾都有一个换行字符，但是我首先没有提到（非常抱歉）*egrep*会在检查正则表达式之前把这些换行符去掉，所以在行尾的q之后，没有能够匹配u以外的字符。

请不要因此灰心丧气（注4）。我向你保证，如果*egrep*保留换行符（许多其他软件会保留这些符号），或者Iraq后紧接着空格或者其他单词，这一行就能匹配。理解工具软件的细节固然很重要，但现在我只希望读者能通过这个例子认识到：**一个字符组，即使是排除型字符组，也需要匹配一个字符。**

在「03.19.76」中，点号是元字符——它能够匹配任意字符（包括我们期望的连字符、句号和斜线）。不过，我们也需要明白，点号可以匹配任何字符，所以这个正则表达式也能够匹配下面的字符串：'lottery numbers: 19 203319 7639'

所以，「03[-./]19[-./]76」更加精确，但是更难读，也更难写。「03.19.76」更容易理解，但是不够细致。我们应该选择哪一个呢？这取决于你对需要检索的文本的了解，以及你需要达到的准确程度。一个重要但常见的问题是，写正则表达式时，我们需要在对欲检索文本的了解程度与检索精确性之间求得平衡。例如，如果我们知道，针对某个检索文本，「03.19.76」这个正则表达式基本不可能匹配不期望的结果，使用它就是合理的。要想正确使用正则表达式，清楚地了解目标文本是非常重要的。

注4：在我四年级的时候，曾经带队参加拼字比赛（spelling bee），题目是拼写"miss"。我的答案是"m·i·s·s"，但史密斯小姐含蓄地告诉我说，应该是"M·i·s·s"，第一个字母要大写，我应该找老师要一个例句，于是我出局了。作为一个小孩子，那时候我感觉非常受伤。从此以后，我开始讨厌史密斯小姐，拼写也学得非常糟糕。

多选结构

Alternation

匹配任意子表达式

「|」是一个非常简捷的元字符，它的意思是**“或”**(or)。依靠它，我们能够把不同的子表达式组合成一个总的表达式，而这个总的表达式又能够匹配任意的子表达式。假如「Bob」和「Robert」是两个表达式，但「Bob|Robert」就是能够同时匹配其中任意一个的正则表达式。在这样的组合中，子表达式称为**“多选分支**(alternative)”。

回头来看「gr[ea]y」的例子，有意思的是，它还可以写作「grey|gray」，或者是「gr(a|e)y」。后者用括号来划定多选结构的范围（正常情况下，括号也是元字符）。请注意，「gr[a|e]y」不符合我们的要求——在这里，‘|’只是一个和「a」与「e」一样的普通字符。

对表达式「gr(a|e)y」来说，括号是必须的，因为如果没有括号，「gra|ey」的意思就成了“「gra」或者「ey」”，而这不符合我们的要求。多选结构可以包括很多字符，但不能超越括号的界限。另一个例子是「(First|1st)·[Ss]treet」（注 5）。事实上，因为「First」和「1st」都以「st」结尾，我们可以把这个结合体缩略表示为「(Fir|1)st·[Ss]treet」。这样可能不容易看得清楚，但我们知道「(First|1st)」与「(fir|1)st」表示的是同一个意思。

下面是一些用多选结构来拼写我名字的例子。这 3 个表达式是一样的，请仔细比较：

「Jeffrey|Jeffery」
「Jeff(rey|ery)」
「Jeff(re|er)y」

英国拼写法如下：

「(Geoff|Jeff)(rey|ery)」
「(Geo|Je)ff(rey|ery)」
「(Geo|Je)ff(re|er)y」

最后要注意的是，这 3 个表达式其实与下面这个更长（但是更简单）的表达式是等价的：「Jeffrey|Geoffery|Jeffery|Geoffrey」。它们只是**“殊途同归”**而已。

「gr[ea]y」与「gr(a|e)y」的例子可能会让人觉得多选结构与字符组没太大的区别，但是请留神不要混淆这两个概念。一个字符组只能匹配目标文本中的**单个字符**，而每个多选结构自身都可能是完整的正则表达式，都可以匹配任意长度的文本。

注 5：请参考第 VI 页的体例说明，这里的“·”表示空格，这样更容易识别。

字符组基本可以算是一门独立的微型语言（例如，对于元字符，它们有自己的规定），而多选结构是“正则表达式语言主体（main regular expression language）”的一部分。你将会发现，这两者都非常有用。

同样，在一个包含多选结构的表达式中使用脱字符和美元符的时候也要小心。比较「^From|Subject|Date:•」和「^(From|Subject|Date):•」就会发现，虽然它们看起来与之前的 E-mail 的例子很相似，匹配结果（即它们的用处）却大不相同。第一个表达式由 3 个多选分支构成，所以它能匹配「^From」或者「Subject」或者「Date:•」，实用性不大。我们希望在每一个多选分支之前都有脱字符，之后都有「:•」。所以应该使用括号来“限制”（constrain）这些多选分支：

「^(From|Subject|Date):•」

现在 3 个多选分支都受括号的限制，所以，这个正则表达式的意思是：匹配一行的起始位置，然后匹配「^From」、「Subject」或「Date」中的任意一个，然后匹配「:•」，所以，它能够匹配的文本是：

1）行起始，然后是 F•r•o•m，然后是 ‘:•’，

或者 2）行起始，然后是 S•u•b•j•e•c•t，然后是 ‘:•’，

或者 3）行起始，然后是 D•a•t•e，然后是 ‘:•’。

简单点说，就是匹配以 ‘From:•’，‘Subject:•’ 或者 ‘Date:•’ 开头的文本行，在提取 E-mail 文件中的信息时这很有用。

下面是一个例子：

```
% egrep '^(From|Subject|Date): ' mailbox
From: elvis@tabloid.org (The King)
Subject: be seein' ya around
Date: Mon, 23 Oct 2006 11:04:13
From: The Prez <president@whitehouse.gov>
Date: Wed, 25 Oct 2006 8:36:24
Subject: now, about your vote…
```

忽略大小写

Ignoring Differences in Capitalization

E-mail header 的例子很适合用来说明**不区分大小写**（case-insensitive）的匹配的概念。E-mail header 中的字段类型（field type）通常是以大写字母开头的，例如“Subject”和“From”，但是 E-mail 标准并没有对大小写进行严格的规定，所以“DATE”或者“from”也是合法的字段类型。但是，之前使用的正则表达式无法处理这种情况。

一种办法是用「[Ff][Rr][Oo][Mm]」取代「From」，这样就能匹配任何形式的“from”，但缺点之一就是很不方便。幸好，我们有一种办法告诉 *egrep* 在比较时忽略大小写，也就是进行不区分大小写的匹配，这样就能忽略大小写字母的差异。

该功能并不是正则表达式语言的一部分，却是许多工具软件提供的有用的相关特性。*egrep* 的命令行参数“-i”表示进行忽略大小写的匹配。把-i 写在正则表达式之前：

```
% egrep -i '^(From|Subject|Date): ' mailbox
```

结果除了包括之前的内容外，还包含这一行：

```
SUBJECT: MAKE MONEY FAST
```

我使用-i 参数的频率很高（也许与第 12 页的注解有关），所以我推荐读者记住它。在下面的章节中我们还会见到其他的简捷特性。

单词分界符

Word Boundaries

使用正则表达式时经常会遇到的一个问题，期望匹配的“单词”包含在另一个单词之中。在 cat、gray 和 Smith 的例子中，我曾提到过这个问题。不过，某些版本的 *egrep* 对单词识别提供了有限的支持：也就是单词分界符（单词开头和结束的位置）的匹配。

如果你的 *egrep* 支持“**元字符序列**（metasequences）”「\<」和「\>」，就可以使用它们来匹配单词分界的位置。可以把它们想象为单词版本的「^」和「$」，分别用来匹配单词的开头和结束位置。就像作为行锚点的脱字符和美元符一样，它们锚定了正则表达式的其他部分，但在匹配过程中并不对应到任何字符。表达式「\<cat\>」的意思是“匹配单词的开头位置，然后是 c·a·t 这 3 个字母，然后是单词的结束位置”。更直接点说就是“匹配 cat 这个单词”。如果读者愿意，也可以用「\<cat」和「cat\>」来匹配以 cat 开头和结束的单词。

请注意，「<」和「>」本身并不是元字符——只有当它们与斜线结合起来的时候，整个序列才具有特殊意义。这就是我称其为“元字符序列”的原因。重要的是它们的特殊意义，而不是字符的个数，所以我说的“元字符”和“元（字符）序列”大多数时候是等价的。

请记住，并不是所有版本的 *egrep* 都支持单词分界符，即使是支持的版本也不见得聪明到能“认得出”英语单词。“单词的起始位置”只不过是一系列字母和数字符号（alphanumeric characters）开始的位置，而“结束位置”就是它们结尾的地方。下一页的图 1-2 说明了一行简单文本中的单词分界符。

（*egrep* 认定的）单词开头位置用向上的箭头标识，单词结束位置用向下的箭头标识。我们看到，“单词的开始和结束”准确地说是“字母数字符号的开始和结束”，不过这样说太麻烦了。

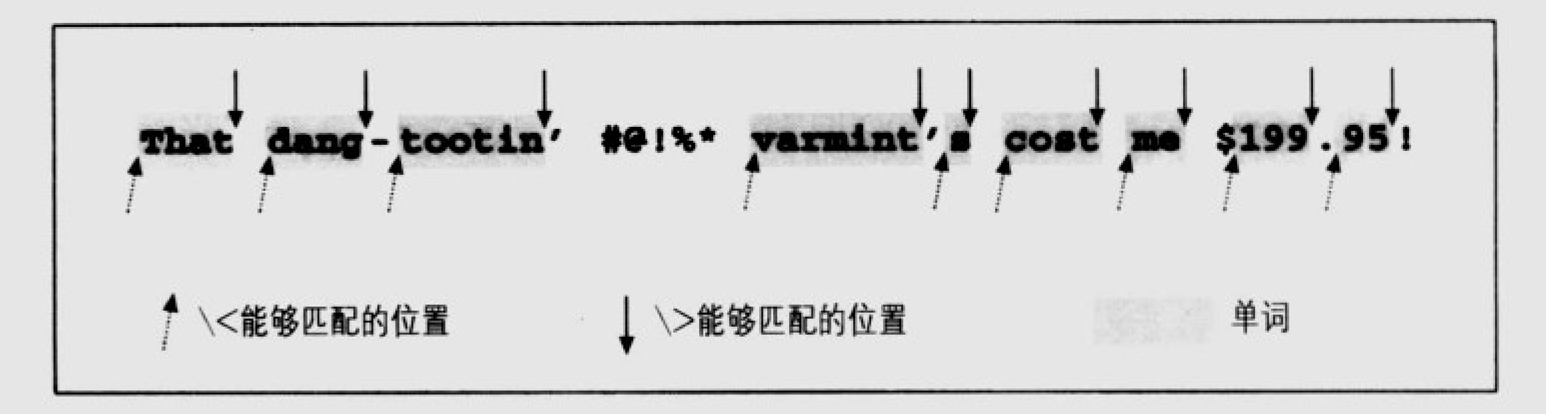


图 1-2："单词"的起始和结束位置

小结

In a Nutshell

表 1-1 总结了我们已经介绍过的元字符。

表 1-1：至今为止所见的元字符小结

元字符	名 称	匹配对象
.	点号	单个任意字符
[···]	字符组	列出的任意字符
[^···]	排除型字符组	未列出的任意字符
^	脱字符	行的起始位置
$	美元符	行的结束位置
\<	反斜线-小于	单词的起始位置(某些版本的 *egrep* 可能不支持)
\>	反斜线-大于	单词的结束位置(某些版本的 *egrep* 可能不支持)
|	竖线	匹配分隔两边的任意一个表达式
(···)	括号	限制竖线的作用范围，其他功能下文讨论

另外还有几点需要注意：

- 在字符组内部，元字符的定义规则（及它们的意义）是不一样的。例如，在字符组外部，点号是元字符，但是在内部则不是如此。相反，连字符只有在字符组内部（这是普遍情况）才是元字符，否则就不是。脱字符在字符组外部表示一个意思，在字符组内部紧接着 [时表示另一个意思，其他情况下又表示别的意思。
- 不要混淆多选项和字符组。字符组「[abc]」和多选项「(a|b|c)」固然表示同一个意思，但是这个例子中的相似性并不能推广开来。无论列出的字符有多少，字符组只能匹配一个字符。相反，多选项可以匹配任意长度的文本，每个多选项可能匹配的文本都是独立的，例如「\<(1,000,000|million|thousand·thou)\>」。不过，多选项没有像字符组那样的排除功能。

- 排除型字符组是表示所有未列出字符的字符组的简便方法。因此,「[^x]」的意思并不是"只有当这个位置不是 x 时才能匹配",而是说"匹配一个不等于 x 的字符"。其中的差别很细微,但很重要。例如,前面的概念可以匹配一个空行,而「[^x]」则不行。
- -i 参数规定在匹配时不区分大小写(☞15)(注 6)。
- 目前介绍过的知识都很有用,但"**可选项**(optional)"和"**计数**(counting)"元素更重要,下文将马上介绍。

可选项元素

Optional Items

现在来看 color 和 colour 的匹配。它们的区别在于,后面的单词比前面的多一个 u,我们可以用「colou**?**r」来解决这个问题。元字符「?」(也就是问号)代表**可选项**。把它加在一个字符的后面,就表示此处容许出现这个字符,不过它的出现并非匹配成功的必要条件。

「u?」这个元字符与我们之前看到的元字符都不相同,它只作用于之前紧邻的元素。因此,「colou?r」的意思是:「c」,然后是「o」,然后是「l」,然后是「o」,然后是「u?」,最后是「r」。

「u?」是必然能够匹配成功的,有时它会匹配一个 u,其他时候则不匹配任何字符。关键在于,无论 u 是否出现,匹配都是成功的。但这并不等于,任何包含?的正则表达式都永远能匹配成功。例如,「colo」和「u?」都能在'semicolon'中匹配成功(前者匹配单词中的 colo,后者什么字符都没有匹配)。可是最后的「r」无法匹配,因此,最终「colou?r」无法匹配 semicolon。

来看另一个例子,我们需要匹配表示 7 月 4 日 (July fourth) 的文本,其中月份可能写作 July 或是 Jul,而日期可能写作 fourth、4th 或者是 4。显然,我们可以使用「(July|Jul)•(fourth|4th|4)」,但也可以找些其他的办法来解决这个问题。

首先,我们把「(July|Jul)」缩短为「(Jul**y?**)」。你明白这种等价变换吗?删除「|」之后,就没必要保留括号了。当然保留也可以,但不保留括号显得更整洁一些。于是我们得到「July?•(fourth|4th|4)」。

注 6:按照第 V 页的体例说明,☞15 代表参照本书第 15 页的内容。

现在来看第二部分，我们可以把「4th|4」简化为「4(th)?」。我们看到，现在「?」作用的元素是整个括号了。括号内的表达式可以任意复杂，但是“从括号外来看”它们是个整体。界定「?」的作用对象（还可以划定我即将介绍的其他类似元字符的作用对象）是括号的主要用途之一。

我们的表达式现在成了「July?·(fourth|4(th)?)」。尽管它包含了许多元字符，而且有嵌套的括号，但理解起来并不困难。我们花了相当的工夫来讲解这两个简单的例子，但同时也接触到了一些相关的知识，它们相当有助于——或许你现在还意识不到——我们理解正则表达式。同样，通过这些讲解，我们也积累了依靠不同思路解决问题的经验。在阅读本书（同时也是在加深理解）寻找复杂问题的最优解决方案的过程中，你可能会发现灵感可能在不断涌现。正则表达式不是死板的教条，它更像是门艺术。

其他量词：重复出现

Other Quantifiers: Repetition

「+」（加号）和「*」（星号）的作用与问号类似。元字符「+」表示“之前紧邻的元素出现一次或多次”，而「*」表示“之前紧邻的元素出现任意多次，或者不出现”。换种说法就是，「…*」表示“匹配尽可能多的次数，如果实在无法匹配，也不要紧”。「…+」的意思与之类似，也是匹配尽可能多的次数，但如果连一次匹配都无法完成，就报告失败。问号、加号和星号这3个元字符，统称为**量词**（quantifiers），因为它们限定了所作用元素的匹配次数。

与「…?」一样，正则表达式中的「…*」也是永远不会匹配失败的，区别只在于它们的匹配结果。而「…+」在无法进行任何一次匹配时，会报告匹配失败。

举例来说，「·?」能够匹配一个可能出现的空格，但是「·*」能够匹配**任意多个**空格。我们可以用这些量词来简化第9页<H[1-6]>的例子。按照HTML规范（注7），在tag结尾的>字符之前，可以出现任意长度的空格，例如<H3·>或者<H4···>。把「·*」加入正则表达式中的可能出现（但不是必须）空格的位置，就得到「<H[1-6]·*>」。它仍然能够匹配<H1>，因为空格并不是必须出现的，但其他形式的tag也能匹配。

注7：即使你不熟悉HTML规范也不必担心。我把它们当作实际的例子，但是我会提供理解其中的意义所需的细节。熟悉HTML tag解析的人可能会认识到我未在本书中提及的重要意义。

接下来看类似<HR·SIZE=14>这样的 HTML tag，它表示一条高度为 14 像素的穿越屏幕的水平线。与<H3>的例子一样，在最后的尖括号之前可以出现任意多个空格。此外，在等号两边也容许出现任意多个空格。最后，在 HR 和 SIZE 之间必须有至少一个空格。为了处理更多的空格，我们可以在「·」后添加「·*」，不过最好还是改写为「·+」。加号确保至少有一个空格出现，所以它与「··*」是完全等价的，只不过更简洁。所以我们得到「<HR·+SIZE·*=·*14·*>」。

尽管这个表达式不受空格数目的限制，但它仍然受 tag 中直线尺寸大小的约束。我们要找的不仅仅是高度为 14 的 tag，而是所有这些 tag。所以，我们必须用能匹配普通数值（general number）的表达式来替换「14」。在这里，“数值”（number）是由一位或多位数字（digits）构成的。「[0-9]」可以匹配一个数字，因为“至少出现一次”，所以我们使用加号量词，结果就是用「[0-9]+」替换「14」。（一个字符组是一个“元素”（unit），所以它可以直接加加号、星号等，而不需要用括号。）

这样我们就得到了「<HR·+SIZE·*=·* [0-9]+·*>」，尽管我用了粗体标识元字符，用空格来分隔各个元素，而且使用了“看得见的空格符”‘·’，这个表达式仍然不容易看懂（幸好，*egrep* 提供了-i 的参数☞15，这样我就不需要用「[Hh][Rr]」来表示「HR」了）。否则，「<HR +SIZE *= *[0-9]+ *>」更令人迷惑。这个表达式之所以看起来有些诡异，是因为星号和加号作用的对象大都是空格，而人眼习惯于把空格和普通字符区分开来。在阅读正则表达式时，我们必须改变这种习惯，因为空格符也是普通字符之一，它与 j 或者 4 这样的字符没有任何差别（在后面的章节中，我们会看到，某些工具软件支持忽略空格的特殊模式）。

我们继续这个例子，如果尺寸这个属性也是可选的，也就是说<HR>就代表默认高度的直线（同样，在 > 之前也可能出现空格）。你能修改我们的正则表达式，让它匹配这两种类型的 tag 吗？解决问题的关键在于明白表示尺寸的文本是**可选**出现的（这是个暗示）。◆**请翻到下一页查看答案。**

请仔细观察最后（答案中）的表达式，体会问号、星号和加号之间的差异，以及它们在实际应用中的真正作用。下一页的表 1-2 总结了它们的意义。

请注意，每个量词都规定了匹配成功至少需要的次数下限，以及尝试匹配的次数上限。对某些量词来说，下限是 0，对某些量词来说，上限是无穷大。

把一个子表达式变为可选项

❖ 19 页问题的答案

在本例中，“可选出现”(optional) 的意思是可以但并不必须匹配一次。这需要使用「?」。因为可选项多于一个字符，所以我们必须使用括号：「(…)?」。把它插入表达式中，就得到「<HR(•+SIZE•*=•*[0-9]+)?•*>」

请注意，结尾的「•*」在「(…)?」以外。这样就能应付「<HR•>」之类的情况。如果我们把它包含在括号内，则只有在出现“SIZE=…”这段文本的情况下才容许在>之前出现空格。

同样请注意 SIZE 之前的「•+」包含在括号内。如果把它拿到括号之外，HR 之后就必须紧跟至少一个空格，即使 SIZE 没有出现也是如此。这样‘<HR>’就无法匹配了。

表 1-2：“表示重复的元字符”含义小结

	次数下限	次数上限	含义
?	无	1	可以不出现，也可以只出现一次（**单次可选**）
*	无	无	可以出现无数次，也可以不出现（**任意次数均可**）
+	1	无	可以出现无数次，但至少要出现一次（**至少一次**）

规定重现次数的范围：区间

某些版本的 *egrep* 能够使用元字符序列来自定义重现次数的区间：「…{*min, max*}」。这称为“**区间**量词 (interval quantifier)”。例如，「…{3,12}」能够容许的重现次数在 3 到 12 之间。有人可能会用「[a-zA-Z]{1,5}」来匹配美国的股票代码（1 到 5 个字母）。问号对应的区间量词是{0,1}。

支持区间表示法的 *egrep* 的版本并不多，但有许多另外的工具支持它。在第 3 章我们会仔细考察目前经常使用的元字符，那时候会涉及区间的支持问题。

括号及反向引用

Parentheses and Backreferences

到目前为止，我们已经见过括号的两种用途：限制多选项的范围；将若干字符组合为一个单元，受问号或星号之类量词的作用。现在我要介绍括号的另一种用途，虽然它在 *egrep* 中并不常见（不过流行的 GNU 版本确实支持这一功能），但在其他工具软件中很常见。

在许多流派（flavor）的正则表达式中，括号能够“记住”它们包含的子表达式匹配的文本。在解决本章开始提到的单词重复问题时就会用到这个功能。如果我们确切知道重复单词的第一个单词（比方说这个单词就是“the”），就能够明确无误地找到它，例如「the•the」。这样或许还是会匹配到 the•theory 的情况，但如果我们的 *egrep* 支持在第 15 页提到的单词分界符「\<the•the\>」，这个问题就很容易解决。我们可以添加「•+」把这个表达式变得更灵活。

然而，穷举所有可能出现的重复单词显然是不可能完成的任务。如果我们先匹配任意一个单词，接下来检查“后面的单词是否与它一样”，就好办多了。如果你的 *egrep* 支持“**反向引用**（backreference）”，就可以这么做。反向引用是正则表达式的特性之一，它容许我们匹配与表达式先前部分匹配的同样的文本。

我们先把「\<the•+the\>」中的第一个「the」替换为能够匹配任意单词的正则表达式「[A-Za-z]+」；然后在两端加上括号（原因见下段）；最后把后一个‘the’替换为特殊的元字符序列「\1」，就得到了「\<**(**[A-Za-z]+**)**•+**\1**\>」。

在支持反向引用的工具软件中，括号能够“记忆”其中的子表达式匹配的文本，不论这些文本是什么，元字符序列「\1」都能记住它们。

当然，在一个表达式中我们可以使用多个括号。再用「\1」、「\2」、「\3」等来表示第一、第二、第三组括号匹配的文本。括号是按照开括号‘(’从左至右的出现顺序进行的，所以「([a-z])([0-9])\1\2」中的「\1」代表「[a-z]」匹配的内容，而「\2」代表「[0-9]」匹配的内容。

在‘the•the’的例子中，「[A-Za-z]+」匹配第一个‘the’。因为这个子表达式在括号中，所以「\1」代表的文本就是‘the’。如果「•+」能够匹配，后面的「\1」要匹配的文本就是‘the’。如果「\1」也能成功匹配，最后的「\>」对应单词的结尾（如果文本是‘the•theft’，这一条就不满足）。如果整个表达式能匹配成功，我们就得到一个重复单词。有的重复单词并不是错误，例如‘that that’（译注 3），这并不是正则表达式的错误，真正的判断还得靠人。

我决定使用上面这个例子的时候，已经用这个表达式检查过本书之前的内容了（我使用的是支持「\<…\>」和反向引用的 *egrep*）。我还使用了第 15 页提到的忽略大小写的参数-i 来拓宽它的适用范围（注 8），所以‘The•the’这样的单词重复也能提取出来。

译注 3：在英文中有可能出现两个连续的 that，例如 And we think that that standard has been met here。其中第一个 that 表示宾语从句的引导词，第二个用于修饰 standards。

注 8：某些版本的 *egrep*，包括一些流行的 GNU 版本的 *egrep* 在内，它们的-i 参数都存在一个 bug，即不会对反向引用的内容忽略大小写，也就是说，它可以找到“the the”，但找不到“The the”。

我使用的命令如下：

```
% egrep -i '\<([a-z]+) +\1\>' files...
```

结果令我惊奇，居然找到了 14 组重复单词。我把它们全都改正了，而且把这个表达式添加到我用来检查本书拼写错误的工具中，保证从此以后全书中不会出现这样的错误。

尽管这个表达式很有用，我们仍然需要重视它的局限。因为 *egrep* 把每行文字都当作一个独立部分来看待，所以如果单词重复的第一个单词在某行末尾，第二个单词在下一行的开头，这个表达式就无法找到。所以，我们需要更加灵活的工具，下一章我们会看到这方面的例子。

神奇的转义

The Great Escape

有个重要的问题我尚未提及，即：如果需要匹配的某个字符本身就是元字符，正则表达式会如何处理呢？例如，如果我想要检索互联网的主机名 `ega.att.com`，使用「`ega.att.com`」可能得到 `megawatt·computing` 的结果。还记得吗？「`.`」本身就是元字符，它可以匹配任何字符，包括空格。

真正匹配文本中点号的元序列应该是反斜线（backslash）加上点号的组合：「`ega\.att\.com`」。「`\.`」称为**"转义的点号"**或者**"转义的句号"**，这样的办法适用于所有的元字符，不过在字符组内部无效（注 9）。

这样使用的反斜线称为"转义符（escape）"——它作用的元字符会失去特殊含义，成了普通字符。如果你愿意，也可以把转义符和它之后的元字符看作特殊的元字符序列，这个元字符序列匹配的是元字符对应的普通字符。这两种看法是等价的。

我们还可以用「`\([a-zA-Z]+\)`」来匹配一个括号内的单词，例如'`(very)`'。在开闭括号之前的反斜线消除了开闭括号的特殊意义，于是他们能够匹配文本中的开闭括号。

如果反斜线后紧跟的不是元字符，反斜线的意义就依程序的版本而定。例如，我们已经知道，某些版本的程序把「`\<`」、「`\>`」、「`\1`」当作元字符序列对待。在后面的章节中我们会看到更多的例子。

注 9：大多数程序设计语言和工具都支持字符组内部的转义，但是大多数版本的 *egrep* 不支持，它们会把反斜线'\'当作字符组内部列出的普通字符。

基础知识拓展

Expanding the Foundation

我希望，前面的例子和解释已经帮助读者牢固地打下了正则表达式的基础，也请读者明白，这些例子都很浅显，我们需要掌握的还有很多。

语言的差异

Linguistic Diversification

我已经介绍过大多数版本的 *egrep* 支持的正则表达式的特性，这样的特性还有很多，其中一些并不是所有的版本都支持，这个问题留到后面的章节讲解。

任何语言中都存在不同的方言和口音，很不幸，正则表达式也一样。情况似乎是，每一种支持正则表达式的语言都提供了自己的“改进”。正则表达式不断发展，但多年的变化也造就了数目众多的正则表达式“流派”（flavor）。我们会在下面的章节中见到各种例子。

正则表达式的目标

The Goal of a Regular Expression

从最宏观的角度看，一个正则表达式要么能够匹配给定文本（对 *egrep* 来说，就是一行文本）中的某些字符，要么不能匹配。在编写正则表达式的时候，我们必须进行权衡：匹配符合要求的文本，同时忽略不符合要求的文本。

尽管 *egrep* 不关心匹配文本在行中的位置，但对正则表达式的其他应用来说，这个问题却很重要。如果文本是这样：

```
…zip is 44272. If you write, send $4.95 to cover postage and…
```

我们只希望找出包含「[0-9]+」的那些行，就不需要关心真正匹配的数字。相反，如果我们需要**操作**这些数字（例如保存到文件、添加、替换之类——我们会在下一章看到这样的处理），就需要关心确切匹配的那些数字。

更多的例子

A Few More Examples

在任何语言中，经验**都是非常重要的**，所以我会给出更多用正则表达式匹配常用文本结构的例子。

编写正则表达式时，按照预期获得成功的匹配要花去一半的工夫，另一半的工夫用来考虑如何忽略那些不符合要求的文本。在实践中，这两方面都非常重要，但是目前我们只关注“获得成功匹配”的方面。即使我没有对这些例子进行最全面彻底的解释，它们仍然能够提供有用的启示。

变量名

许多程序设计语言都有标识符（identifier，例如变量名）的概念，标识符只包含字母、数字以及下画线，但不能以数字开头。我们可以用「[a-zA-Z_][a-zA-Z_0-9]*」来匹配标识符。第一个字符组匹配可能出现的第一个字符，第二个（包括对应的「*」）匹配余下的字符。如果标识符的长度有限制，例如最长只能是32个字符，又能使用第20页介绍的区间量词「{*min*, *max*}」，我们可以用「{0,31}」来替代最后的「*」。

引号内的字符串

匹配引号内的字符串最简单的办法是使用这个表达式：「"[^"]*"」。

两端的引号用来匹配字符串开头和结尾的引号。在这两个引号之间的文本可以包括双引号之外的任何字符。所以我们用「[^"]」来匹配除双引号之外的任何字符，用「*」来表示两个引号之间可以存在任意数目的非双引号字符。

关于引号字符串，更有用（也更复杂）的定义是，两端的双引号之间可以出现由反斜线转义的双引号，例如"nail·the·2\"x4\"·plank"。在后面的章节讲解匹配实际进行的细节时，我们会多次遇到这个例子。

美元金额（可能包含小数）

「\$[0-9]+(\.[0-9][0-9])?」是一种匹配美元金额的办法。

从整体上看，这个表达式很简单，分为三部分：「\$」、「…+」和「(…)?」，可以大致理解为：一个美元符号，然后是一组字符，最后可能还有另一组字符。这里的“字符”指的是数字（一组数字构成一个数值），“另一组字符”是由一个小数点和两位数字构成的。

从几个方面来看，这个表达式还很简陋。比如，它只能接受$1000，而无法接受$1,000。它确实能接受可能出现的小数部分，但对于 *egrep* 来说意义不大。因为 *egrep* 从不关心匹配文字的内容，而只关心是否存在匹配。处理可能出现的小数部分对整个表达式能否匹配并没有影响。

但是，如果我们需要找到只包含价格而不含其他字符的行，倒是可以在这个表达式两端加上「^…$」。这样一来，可选的小数部分就变得很重要了，因为在金额数值和换行符之间是否存在小数部分，决定了整个表达式的匹配结果是否存在差异。

另外，这个正则表达式还无法匹配‘$.49’。你可能认为把加号换成星号能够解决问题，不过这条路走不通。在这我先卖个关子，答案留待第5章（☞194）揭晓。

HTTP/HTML URL

Web URL 的形式可能有很多种，所以构造一个能够匹配所有形式的 URL 的正则表达式颇有难度。不过，稍微降低一点要求的话，我们能够用一个相当简单的正则表达式来匹配大多数常见的 URL。进行这种检索的原因之一是，我只能大概记得在收到的某封邮件中有一个 URL 地址，不过一见到它我就能认出来。

常见的 HTTP/HTML URL 是下面这样的：

http://hostname/path.html

当然，.htm的结尾也很常见。

hostname（主机名，例如 *www.yahoo.com*）的规则比较复杂，但是我们知道，跟在‘http://’之后的就有可能是主机名，所以这个正则表达式就很简单，「[-a-z0-9_.]+」。*path* 部分的变化更多，所以我们需要使用「[-a-z0-9_:@&?=+,.!/~*%$]*」。请注意，连字符必须放在字符组的开头，保证它是一个普通字符，而不是用来表示范围（☞9）。

综合起来，我们第一次尝试的正则表达式就是：

```
% egrep -i '\<http://[-a-z0-9_.:]+/[-a-z0-9_:@&?=+,.!/~*%$]*\.html?\>'
files
```

因为我们降低了对匹配的要求，所以‘http://..../foo.html’也能匹配，虽然它显然不是一个合法的 URL。我们需要关心这一点吗？这取决于具体的情况。如果我只是需要扫描自己的 E-mail，得到一些错误结果并不算是问题。而且，我没准会用更简单的表达式：

```
% egrep -i '\<http://[^ ]*\.html?\>' files...
```

在深入了解如何调校正则表达式之后，读者会明白，要想在复杂性和完整性之间求得平衡，一个重要的因素是了解待搜索的文本。下一章，我们会更详细地考察这个例子。

HTML tag

对 *egrep* 这样的工具来说，简单地匹配包含 HTML tag 的行并不常见，也没什么用。但是，探索如何准确匹配一个 HTML tag 却是相当有启发的，在下一章深入接触更高级的工具时，这一点尤其明显。

简单的例子包括 ‘<TITLE>’ 和 ‘<HR>’，我们可能会想到「<.*>」。这个简单的表达式往往是最直接的想法，但它显然是不对的。「<.*>」的意思是，“先匹配一个 ‘<’，然后是任意多个任意字符，然后是‘>’”。所以，它无疑能够匹配不止一个 tag 的内容，例如‘this <I>short</I> example’ 中标记的内容。

也许结果有点出乎你的意料，但是我们目前还只在第 1 章，对正则表达式的理解也不够深入。我之所以举这个例子，是想说明正则表达式并不复杂，但是如果你不真正弄懂它们，可能会被搞得晕头转向。在下面的几章中，我们会学习理解和解决这个问题需要的所有细节。

表示时刻的文字，例如 “9:17 am” 或者 “12:30 pm”

匹配表示时刻的文字可能有不同的严格程度。

「[0-9]?[0-9]:[0-9][0-9]•(am|pm)」

能够匹配 9:17•am 或者 12:30•pm，但也能匹配无意义的时刻，如 99:99•pm。

首先看小时数，我们知道，如果小时数是一个两位数，第一位只能是 1。但是「1?[0-9]」仍然能够匹配 19（也能够匹配 0），所以更好的办法应该是把小时部分分为两种情况来处理，「1[012]」匹配两位数，「[1-9]」匹配一位数，结果就是「(1[012]|[1-9])」。

分钟数就简单些。第一位数字应该是「[0-5]」，此时第二位数字应该是「[0-9]」。综合起来就是「(1[012]|[1-9]):[0-5][0-9]•(am|pm)」。

举一反三，你能够处理 24 小时制的时间吗？多动动脑筋，想想该如何处理以 0 开头的情况，比如 09:59 呢？◆答案请见下页。

正则表达式术语汇总

Regular Expression Nomenclature

正则（regex）

你或许已经猜到了，“正则表达式”（regular expression）这个全名念起来有点麻烦，写出来就更麻烦。所以，我一般会采用“正则”（regex）的说法。这个单词念起来很流畅（有点像联邦快递的 FedEx，与 regular 一样，g 发重音，而不同于 Regina），而且说“如果你写一个正则”，“巧妙的正则”（budding regexers），甚至是“正则化”（regexification）（注 10）（译注 4）。

匹配（matching）

一个正则表达式“匹配”一个字符串，其实是指这个正则表达式能在字符串中找到匹配文本。严格地说，正则表达式「a」不能**匹配** `cat`，但是能匹配 `cat` 中的 `a`。几乎没人会混淆这两个概念，但澄清一下还是有必要的。

元字符（metacharacter）

一个字符是否元字符（或者是“元字符序列”（metasequence），这两个概念是相等的），取决于应用的具体情况。例如，只有在字符组外部并且是在未转义的情况下，「*」才是一个元字符。“转义”（escaped）的意思是，通常情况下在这个字符之前有一个反斜线。「*」是对「*」的转义，而「*」则不是（第一个反斜线用来转义第二个反斜线），虽然在两个例子中，星号之前都有一个反斜线。

正则表达式的流派（flavor）不同，关于字符转义的规定也不相同。第 3 章对此进行了详细讨论。

流派（flavor）

我已经说过，不同的工具使用不同的正则表达式完成不同的任务，每样工具支持的元字符和其他特性各有不同。我们再举单词分界符的例子。某些版本的 *egrep* 支持我们曾见过的 `\<…\>`表示法。而另一些版本不支持单独的起始和结束边界，只提供了统一的「\b」元字符（这个元字符我们还没见过，下一章才会用到）。还有些工具同时支持这两种表示法，另有许多工具哪种也不支持。

我用“流派（flavor）”这个词来描述所有这些细微的实现规定。这就好像不同的人说不同的方言一样。从表面上看，“流派”指的是关于元字符的规定，但它的内容远远不止这些。

注 10：你或许会想到那个难看的缩写 regexp。我不知道这个词应该如何发音，口齿不清的人读起来可能会容易一点。

译注 4：正则表达式的全名是 regular expression，简写为 regex，原著中此处的标题即为 regex，采用中文简称“正则”。

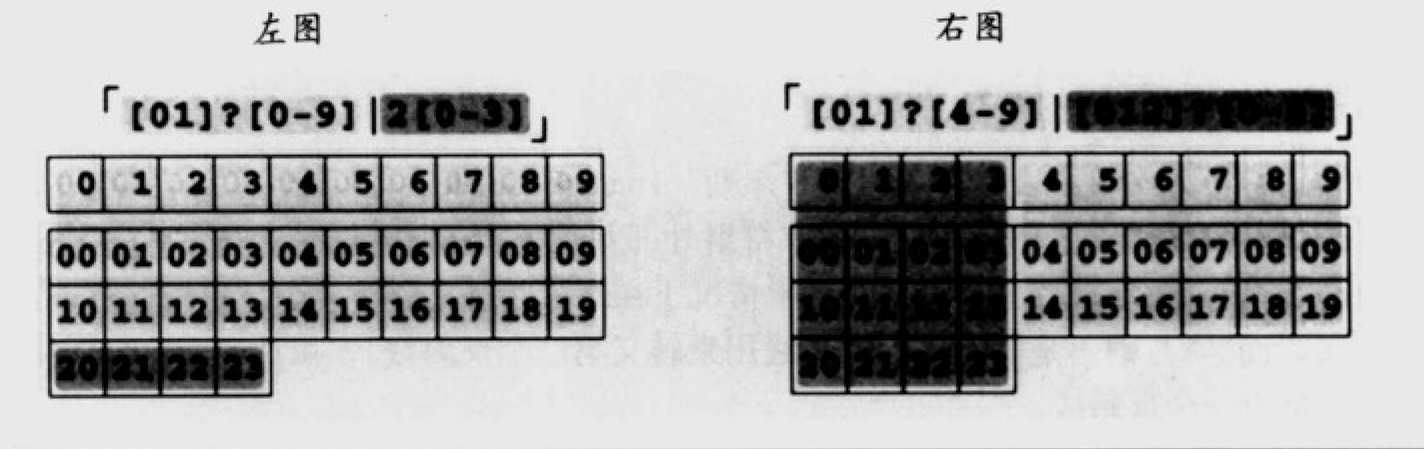
即使两个程序都支持「\<…\>」，它们可能对这两个元字符的意义有不同的理解，对单词的理解也不相同。在使用具体的工具软件时，这个问题尤其重要。

改进匹配时间的表达式，处理 24 小时制时间

❖ 26 页问题的答案

办法有许多种，不过思路和之前差不多。现在我们把问题分为 3 部分：其一是上午（小时数从 00 到 09，开头的 0 可选），其二是白天（小时数从 10 到 19），其三是夜晚（小时数从 20 到 23）。这样答案就很明显了：「0?[0-9]|1[0-9]|2[0-3]」。

实际上，我们可以合并头两个多选分支，得到「[01]?[0-9]|2[0-3]」。你可能需要动点脑筋才能明白这个表达式与上面是完全等价的。下面的图可能有所帮助，它还提供了另一种思路。阴影部分表示单个多选分支能够匹配的数字。



请不要混淆“流派（flavor）”和“工具（tool）”这两个概念。两个人可以说同样的方言，两个完全不同的程序也可能属于同样的流派。同样，两个名字相同的程序（解决的任务也相同）所属的流派可能有细微（有时可能并非细微）的差别。有许多程序都叫 *egrep*，它们所属的流派也五花八门。

由 Perl 语言的正则表达式开创的流派，在 20 世纪 90 年代中期因为其强大的表达能力广为人们所知，其他语言紧随其后，提供了汲取其中灵感的正则表达式（其中许多为了标明自己的思想来源，直接给自己贴上“兼容 Perl（Perl-Compatible）”的标签）。它们包括 PHP、Python、Java 的大量正则包， 微软的.NET Framework、Tcl，以及 C 的各种类库。不过，所有这些语言在重要的方面各有不同。而且 Perl 的正则表达式也在不断演化和发展（现在，有时候是受了其他语言的正则表达式的刺激）。像往常一样，总的局面变得越来越复杂，让人困惑。

子表达式（subexpression）

“子表达式”指的是整个正则表达式中的一部分，通常是括号内的表达式，或者是由「|」分隔的多选分支。例如，在「^(Subject|Date):·」中，「Subject|Date」通常被视为一个子表达式。其中的「Subject」和「Date」也算得上子表达式。而且，严格说起来，「S」、「u」、「b」、「j」这些字符，都算子表达式。

1-6 这样的字符序列并不能算「H[1-6]·*」的子表达式，因为‘1-6’所属的字符组是不可分割的“单元（unit）”。但是，「H」、「[1-6]」、「·*」都是「H[1-6]·*」的子表达式。

与多选分支不同的是，量词（星号、加号和问号）作用的对象是它们之前紧邻的子表达式。所以「mis+pell」中的 + 作用的是「s」，而不是「mis」或者「is」。当然，如果量词之前紧邻的是一个括号包围的子表达式，整个子表达式（无论多复杂）都被视为一个单元。

字符（character）

“字符”在计算机领域是一个有特殊意义的单词。一个字节所代表的单词取决于计算机如何解释。单个字节的值不会变化，但这个**值**所代表的**字符**却是由解释所用的编码来决定的。例如，值为 64 和 53 的字节，在 ASCII 编码中分别代表了字符“@”和“5”，但在 EBCDIC 编码中，则是完全不同的字符（一个是空格，一个是控制字符）。

另一方面，在流行的日文字符编码中，这两个字节代表一个字符正。如果换一种日文字符编码，这个字就需要两个完全不同的字节。那两个字节，在通行的 Latin-1 编码中，表示“Àµ”，而在 Unicode 编码中又表示韩文的“솨”（注 11）。问题在于，字节如何解释只是视角（称为“**编码**”encoding）的问题，我们要做的只是确保自己的视角和正在使用的工具的视角相同。

注 11：关于多字节编码的权威著作是 Ken Lunde 的 *CJKV Information Processing*，这本书亦由 O'Reilly 出版。CJKV 表示 *Chinese*、*Japanese*、*Korean* 和 *Vietnamese*（汉语、日语、韩语和越南语），这 4 种语言都必须使用多字节编码。Ken 和 Adobe 热心地提供了本书所需的特殊字体。

一直以来，文本处理软件一般都把数据视为一些 ASCII 编码的字节，而不考虑使用者期望采用的字符编码。不过，近来已经有越来越多的系统在内部使用某些格式的 Unicode 编码来处理数据（第 3 章介绍了 Unicode，☞105）。如果这些系统中的正则表达式子系统的实现方式正确，使用者通常就不需要在编码的问题上费太多工夫。这个“如果”相当复杂，所以第 3 章深入讲解了这个问题。

改进现状

Improving on the Status Quo

总的来说，正则表达式并不难。但是，如果你与使用过支持正则表达式的程序或语言的人交流过就会发现，某些人确实“会用”正则表达式，但如果需要解决复杂的问题，或是换用他们不熟悉的工具，就会出问题。

传统的正则表达式文档大都只包含一两个元字符的简略介绍，然后就给出关于其他元字符的表格。给出的例子通常也是无意义的「a*((ab)*|b*)」，文本则是‘a·xxx·ce·xxxxxx·ci·xxx·d’。这些文档大都忽略了细微但重要的知识点，总是声称自己与其他出名的工具属于同一流派，而忘记提及必然存在的差异。它们缺乏实用价值。

当然，我的意思并不是，本章就能够填补这道鸿沟，让读者掌握所有正则表达式，或是掌握 *egrep* 的正则表达式。相反，这一章只是为本书的其他内容铺垫基础。我希望本书能够为读者填补这道鸿沟，虽然这期望有点自负。很多读者很满意本书的第一版，我本人也为拓展这一版的深度和广度付出了艰苦的努力。

或许是因为正则表达式的文档一直都非常欠缺，我感到自己必须做出额外的努力，才能把知识梳理清楚。因为我希望保证读者能够充分运用正则表达式的潜力，我希望你们能够**真正**精通正则表达式。

这既是件好事也是件坏事。

好处在于，你将学会如何以正则表达式的方式来**思考**问题。你将学习到，在面对属于不同流派的新工具时，需要注意哪些差异和特性。你还将会学习到，如果某个流派的功能弱小、特性简陋，该如何表达自己的意图。你将会明白，一个正则表达式的效率优于其他表达式的原因所在，而且你将能够在复杂性、效率和匹配准确性间进行取舍权衡。

面对特别复杂的任务，你将会知道如何通过程序容许的方式来构建和使用正则表达式。总的来说，你能够得心应手地使用正则表达式的所有潜能。

问题在于，这种方法的学习曲线非常陡峭，而且还有几大难点：

- **正则表达式的使用** 许多程序使用的正则表达式比 *egrep* 要复杂。在我们探讨如何构造真正有用的正则表达式的细节之前，需要知道正则表达式的使用方法。下一章关注这一问题。
- **正则表达式的特性**（feature） 面对问题，选择合适的工具是成功的一半，所以我会在全书中使用多种工具。不同的程序，甚至是同一个程序的不同版本，支持的特性和元字符都不一样。在了解使用细节之前，我们必须搞清楚这个问题。这是第 3 章的主题。
- **正则表达式的工作原理** 在我们接触有用（但通常也很复杂）的例子之前，我们必须“揭开盖子”来了解正则表达式的工作原理。我们将会看到，对某些元字符进行尝试匹配的次序是一个重要的问题。实际上，正则表达式引擎（regular expression engine）不同，工作原理也不同，所以对于同样的正则表达式，不同的程序会得到不同的结果。我们将在第 4、5、6 章中探讨这个复杂的问题。

正则表达式的工作原理是最重要同时也是最难以掌握的知识。研究这个问题有时的确很枯燥，更糟糕的是，读者在接触真正有趣的内容——解决实际问题——之前，不得不耐着性子看完它们。然而，弄懂正则表达式的工作原理，才是**真正理解**的关键。

你或许会想，如果只希望学会开车，是不需要了解汽车运行原理的。但是，学习开车与学习正则表达式之间并没有多少相似性。我的目的是教会读者如何使用正则表达式——也就是编写正则表达式——来解决问题。更合适的比喻是，学习正则表达式就如同学习如何造车，而不是如何开车。在制造汽车以前，我们必须了解汽车的工作原理。

第 2 章提供了更多的关于开车的经验。第 3 章简要回顾了开车的历史，详细考察了正则表达式流派的主要内容。第 4 章介绍了正则表达式流派的重要的引擎。第 5 章展示了一些更复杂的例子，第 6 章告诉你如何调校某种具体的引擎，之后的各章则是检查具体的产品和模型。在第 4、5、6 章中，我们花了大量的篇幅来探讨幕后的原理，所以请务必做好准备。

总结

Summary

表 1-3 总结了我们在本章中见过的 *egrep* 的元字符。

表 1-3：egrep 的元字符总结

匹配单个字符的元字符		
	元字符	匹配对象
.	点号	匹配单个任意字符
[⋯]	字符组	匹配单个列出的字符
[^⋯]	排除型字符组	匹配单个未列出的字符
char	转义字符	若 *char* 是元字符，或转义序列无特殊含义时，匹配 *char* 对应的普通字符
提供计数功能的元字符		
?	问号	容许匹配一次，但非必须
*	星号	可以匹配任意多次，也可能不匹配
+	加号	至少需要匹配一次，至多可能任意多次
{*min*, *max*}	区间量词①	至少需要 *min* 次，至多容许 *max* 次
匹配位置的元字符		
^	脱字符	匹配一行的开头位置
$	美元符	匹配一行的结束位置
\\<	单词分界符①	匹配单词的开始位置
\\>	单词分界符①	匹配单词的结束位置
其他元字符		
|	alternation	匹配任意分隔的表达式
(⋯)	括号	限定多选结构的范围，标注量词作用的元素，为反向引用"捕获"文本
1, *2*, ...	反向引用①	匹配之前的第一、第二组括号内的字表达式匹配的文本

①并非所有版本的 *egrep* 都支持

此外，请务必理解以下几点：

- 各个 *egrep* 程序是有差别的。它们支持的元字符，以及这些元字符的确切含义，通常都有差别——请参考相应的文档（☞23）。
- 使用括号的 3 个理由是：限制多选结构（☞13）、分组（☞14）和捕获文本（☞21）。
- 字符组的特殊性在于，关于元字符的规定是完全独立于正则表达式语言"主体"的。

- 多选结构和字符组是截然不同的，它们的功能完全不同，只是在有限的情况下，它们的表现相同（☞13）。
- 排除型字符组同样是一种“肯定断言”（positive assertion）——即使它的名字里包含了“排除”两个字，它仍然需要匹配一个字符。只是因为列出的字符都会被排除，所以最终匹配的字符肯定不在列出的字符之内（☞12）。
- -i 的参数很有用，它能进行忽略大小写的匹配（☞15）。
- 转义有3种情况：

  1. 「\」加上元字符，表示匹配元字符所使用的普通字符（例如「*」匹配普通的星号）。
  2. 「\」加上非元字符，组成一种由具体实现方式规定其意义的元字符序列（例如，「\<」表示“单词的起始边界”）。
  3. 「\」加上任意其他字符，默认情况就是匹配此字符（也就是说，反斜线被忽略了）。

  请记住，对大多数版本的 *egrep* 来说，字符组内部的反斜线没有任何特殊意义，所以此时它并不是一个转义字符。
- 由星号和问号限定的对象在“匹配成功”时可能并没有匹配任何字符。即使什么字符都不能匹配到，它们仍然会报告“匹配成功”。

一家之言

Personal Glimpses

本章开始的单词重复的例子可能有些让人迷惑，不过正则表达式的功能的确很强大，我们只需要 *egrep* 这样简单的工具，用第1章的知识，就能够基本解决这个问题。我倒是希望在本章多讲些复杂点的例子，但是因为我希望用第1章为后面的章节打下坚实的基础，我担心，如果这一章满是提醒、注意、规则之类，那些从未接触过正则表达式的人在读完之后，或许会感到困惑——“需要这么麻烦吗？”

我哥哥曾教朋友玩一种叫 *schaffkopf* 的纸牌游戏，这个游戏在我们家流传了好几代了。其中真正的乐趣并不会在第一次接触时体会到，而且学习的曲线也很陡峭。我的嫂子丽兹平时是最有耐心的人，但玩了半个小时就对这些复杂的规则感到灰心丧气了，她说“我们不能不能玩兰米（译注5）吗？”不过最后的结果是，包括丽兹在内的所有人都玩到很晚。只要

译注5：兰米 rummy 是一种比较简单、流行的玩法。

度过了开始的难关，接下来的刺激就会引诱他们继续向前。我哥哥知道肯定会这样，但丽兹和其他玩伴需要花费时间和工夫来继续深入才能体会到这个游戏的乐趣。

习惯正则表达式可能需要花一些时间，所以在你没有真切体会到用正则表达式解决问题的成就感时，可能会觉得这样有点迂腐。如果是，我希望你能够抵抗住“玩兰米游戏”的诱惑。一旦你掌握了正则表达式的强大功能，就会感到花在学习上的那些时间真是物超所值。



第2章 入门示例拓展

Extended Introductory Examples

还记得第 1 章中单词重复的例子吗？我说过，完整解决这个问题只需要用 Perl 之类的语言写几行代码。它看起来像是这样：

```
$/ = ".\n";
    while (<>) {
      next if !s/\b([a-z]+)((?:\s|<[^>]+>)+)(\1\b)/\e[7m$1\e[m$2\e[7m$3\e[m/ig;
      s/^(?:[^\e]*\n)+//mg;    # 去除未标记的行
      s/^/$ARGV: /mg;          # 在行首添加文件名
      print;
    }
```

嗯哼，这就是完整的程序了。

即便你对 Perl 有所了解，我也不敢奢望你能完全明白这段程序（至少目前如此）。我希望的是，这个例子让你看到 *egrep* 之外的世界，让你有兴趣认识正则表达式的真正能力。

该程序的主要功能依靠 3 个正则表达式：

- 「\b([a-z]+)((?:\s|<[^>]+>)+)(\1\b)」
- 「^(?:[^\e]*\n)+」
- 「^」

尽管这是一个 Perl 的例子，但这 3 个正则表达式可以原封不动地（或者只需要做很少的改动）应用到许多其他语言中，比如 PHP、Python、Java、VB.NET、Tcl 等等。

现在来看这 3 个表达式，最后的「^」很好理解，但是其他的两个表达式包含我们在 *egrep* 中未见过的玩意儿。这是因为 Perl 与 *egrep* 不属于同一个流派，所以某些表示法有所不同，而且 Perl（还包括许多其他现代的工具程序）提供的元字符远远多于 *egrep*。我们会在这一章中见到许多例子。

关于这些例子

About the Examples

本章包括了一些常见的问题——验证用户的输入数据，处理 E-mail header（电子邮件头），把纯文本数据转换为超文本格式（HTML），通过这些问题，你将真正见识到正则表达式的世界。在构造正则表达式时，我会做些尽可能详细的讲解，提供一些启示。在这个过程中，我们会见到一些 *egrep* 没有提供的结构和特性，也会专门花很多篇幅来探讨其他重要的概念。

在本章的末尾及下面的各章中，我会使用各种语言，包括 PHP、Java 和 VB.NET，但是本章中我们主要使用的还是 Perl。这些语言，当然也包括其他许多语言，对正则表达式的操纵能力都远远强于 *egrep*，所以使用其中任何一种作为示范都会让我们见到许多有趣的内容。我选择以 Perl 开始，主要是因为，在所有流行的语言中 Perl 对正则表达式的支持很完整，且易于使用。而且，Per 还提供了许多其他紧凑的数据处理结构（data-handling constructs），能够减少所需的“简单重复劳动”（dirty work），以便我们把精力集中到正则表达式上。

第 2 页出现的文件检查的程序很好地说明了这种能力，我需要用这个程序来确定每个文件中‘`ResetSize`’出现的次数与‘`SetSize`’出现的次数是否一样多。我选择的语言是 Perl，命令如下：

```
% perl -0ne 'print "$ARGV\n" if s/ResetSize//ig != s/SetSize//ig' *
```

（我并不奢望你现在就能明白这条命令——我只希望你能注意到这条命令有多简洁。）

我喜欢 Perl，但现在在讨论的主题不是 Perl。请记住，本章的重点是**正则表达式**。这有点儿像计算机系的教授在第一学年的课堂上说的“你们将会在这里学习计算机科学知识，但我们选择用 Pascal 语言作为学习的工具”（注 1）。

因为本章并不假设读者已经懂得 Perl，所以我会做些必要的讲解，让你明白这些例子（第 7 章讲解 Perl 的本质的细节，它假设读者懂得一些基本的知识）。即使你曾经用过好几门语言，Perl 也可能让你觉得奇怪，因为它的语法极精炼，语意又极丰富。为了让这些例子更清楚，我不会使用 Perl 提供的这些特性，而是用一种更普通的近乎伪码的风格来展示这些程序。虽然算不上“蹩脚”，但这些例子也不符合 Perl 的编程风格。不过，我们将通过它们认识到正则表达式的重要作用。

注 1：Pascal 是一种传统的程序设计语言，设计的初衷是为了教学。这个比喻来自 William F.Maton 和他的教授。

Perl 简单入门

A Short Introduction to Perl

Perl 是一门功能强大的脚本语言，诞生于 20 世纪 80 年代末期，其思想主要来自其他的编程语言和工具。Perl 关于文本处理和正则表达式的许多概念来自两种专业化的语言 *awk* 和 *sed*，它们都非常不同于“传统”的语言，例如 C 和 Pascal。

Perl 可以应用于许多平台，包括 DOS/Windows、MacOS、OS/2、VMS 和 Unix。它的文本处理能力极其强大，是关于 Web 的处理中最常使用的工具。如果要获得对应你的机器版本的 Perl，请参考 *www.perl.com*。

本书是为 Perl 的 5.8 版而写的，不过本章的例子可以在 5.005 以后的版本上使用。

现在来看一个简单的例子：

```
$celsius = 30;
$fahrenheit = ($celsius * 9 / 5) + 32;   # 计算华氏温度
print "$celsius C is $fahrenheit F.\n";  # 返回摄氏和华氏温度
```

执行这段程序，结果是：

```
30 C is 86 F.
```

`$fahrenheit` 和`$celsius` 之类的普通变量一般以`$`开头，可以保存一个数值或者任意长度的文本（在本例中只保存了数值）。从`#`到行尾都是注释。

如果你曾经使用过 C、C#或者 Java、VB.NET，你可能无法理解在 Perl 中变量居然能够出现在双引号包围的字符串中。在字符串“`$celsius C is $fahrenheit F.\n`”中，每个变量都会被它的实际值所取代。在本例中，结果就是打印出来的字符串（\n 代表换行）。

Perl 也提供了跟其他流行的语言类似的控制结构：

```
$celsius = 20;
while ($celsius <= 45)
{
  $fahrenheit = ($celsius * 9 / 5) + 32;   # 计算华氏温度
  print "$celsius C is $fahrenheit F.\n";
  $celsius = $celsius + 5;
}
```

在条件为真（即**`$celsius <= 45`**）时，`while` 循环控制的部分会重复执行。把这段代码写入到一个程序中，例如 *temps*，我们可以直接从命令行运行它。

下面是运行的结果：

```
% perl -w temps
20 C is 68 F.
25 C is 77 F.
30 C is 86 F.
35 C is 95 F.
40 C is 104 F.
45 C is 113 F.
```

-w 参数并不是运行所必须的，与正则表达式也没有直接的联系。它只是告诉 Perl，仔细检查我们的程序，在 Perl 认为可疑的地方发出警报（例如没有初始化的变量之类——在 Perl 中，变量不需要事先声明就能使用）。在这里加上这个参数，只是因为它是一种良好的习惯。

好了，这就是 Perl 的简单入门。下面我们来看在 Perl 中如何使用正则表达式。

使用正则表达式匹配文本

Matching Text with Regular Expressions

Perl 可以以多种方式使用正则表达式，最简单的就是检查变量中的文本能否由某个正则表达式匹配。下面的代码检查$reply 中所含的字符串，报告这个字符串是否全部由数字构成：

```
if ($reply =~ m/^[0-9]+$/) {
   print "only digits\n";
} else {
   print "not only digits\n";
}
```

第一行的代码也许有些奇怪：正则表达式是「^[0-9]+$」，两边的 **m/…/**告诉 Perl 该对这个正则表达式进行什么操作。**m** 代表尝试进行 **"正则表达式匹配**（regular expression match）"，斜线用来标记界限(注 2)。之前的**=~**用来连接 m/…/和欲搜索的字符串，即本例中的$reply。

请不要混淆**=~**、**=**和**==**。运算符**==**用来测试两个数字是否相等（我们将会看到，运算符 **eq** 用来测试两个**字符串**是否相等）。运算符**=**用来给变量赋值，例如**$Celsius = 20**。最后，**=~** 用来连接正则表达式和待搜索的目标字符串。在这个例子中，要搜索的正则表达式是 m/^[0-9]+$/，而目标字符串是$reply。此程序在其他语言中的思路有所不同，我们会在下一章看到例子。

注 2：在许多情况下，m 并不是必须出现的。这个例子用**$reply =~ /^[0-9]+$/**也可以，有过 Perl 编程经验的读者或许会觉得这样更顺眼。我个人认为，加上 m 更清晰，所以我喜欢这么做。

把=~读作“匹配（matches）”可能比较省事，所以

```
if ($reply =~ m/^[0-9]+$/)
```

读作：

“如果变量$reply所含的文本**能够匹配**正则表达式「^[0-9]+$」，那么…”

如果「^[0-9]+$」能够匹配$reply的内容，**$reply =~ m/^[0-9]+$/**的返回值就为*true*，否则为*false*。if语句使用true/false值来决定输出什么信息。

请注意，如果$reply中包含**任意的**数字字符，**$reply =~ m/[0-9]+/**（相比之前的表达式，去掉了开头的脱字符和结尾的美元符）的返回值就是true。两端的「^…$」保证整个$reply**只包含**数字。

现在把上面两个例子结合起来。首先提示用户输入一个值，接收这个输入，用一个正则表达式来验证，确保输入的是一个数值。如果是，我们就计算相应的华氏温度，否则，我们输出一条报警信息：

```
print "Enter a temperature in Celsius:\n";
$celsius = <STDIN>; # 从用户处接受一个输入
chomp($celsius);     # 去掉$celsius后面的换行符

if ( $celsius =~ m/^[0-9]+$/) {
    $fahrenheit = ($celsius * 9 / 5) + 32; #计算华氏温度
    print "$celsius C is $fahrenheit F\n";
} else {
    print "Expecting a number, so I don't understand \"$celsius\".\n";
}
```

请注意最后的print语句有两个转义的双引号，它们的作用并不是标记引用字符串的边界。对大多数语言的文字字符串（literal string）来说，有时候需要转义某些字符，做法跟正则表达式中元字符的转义很相似。在Perl中，字符串与正则表达式的区别并非很重要，但是在Java、Python等语言中却极为重要。“一点题外话——数量丰富的元字符”这一节（☞44）更详细地讨论了这个问题（VB.NET是个明显的例外，在那里转义双引号用‘""’而不是‘\"’）。

如果我们把这段程序保存为*c2f*，则运行结果如下：

```
% perl -w c2f
Enter a temperature in Celsius:
22
22 C is 71.599999999999994316 F
```

哎呀，看来（至少在某些系统上），Perl的简单的print并不能很好地处理浮点数。

我不想在本章中讨论 Perl 的细节，但是我告诉你用 printf（"格式化输出（print formatted）"）可以解决这个问题：

```
printf "%.2f C is %.2f F\n", $celsius, $fahrenheit;
```

这里的 printf 类似 C 语言中的 printf，或者 Pascal、Tcl、*elisp* 和 Python 中的 format。它不会更改变量的值，而只是改变显示的方式。现在的结果好看多了：

```
Enter a temperature in Celsius:
22
22.00 C is 71.60 F
```

向更实用的程序前进

Toward a More Real-World Example

让我们扩展这个例子，容许输入负数和可能出现的小数部分。这个问题的计算部分没问题——Perl 通常情况下不区分整数和浮点数。不过我们需要修改正则表达式，容许输入负数和浮点数。我们添加一个「-?」来容许最前面的负数符号。实际上，我们可以用「[-+]?」来处理开头的正负号。

要容许可能出现的小数部分，我们添加「(\.[0-9]*)?」。转义的点号匹配小数点，所以「\.[0-9]*」用来匹配小数点和后面可能出现的数字。因为「\.[0-9]*」被「(…)?」所包围，整个子表达式都不是匹配成果所必须的（请注意，它与「\.?[0-9]*」是截然不同的，对后一个表达式中，即使「\.」无法匹配，「[0-9]*」也能够匹配接下来的数字）。

把这些综合起来，就得到这样的条件判断语句：

```
if ($celsius =~ m/^[-+]?[0-9]+(\.[0-9]*)?$/) {
```

它能够匹配 32、-3.723、+98.6 这样的文字。不过还不够完善：它不能匹配以小数点开头的数（例如.357）。当然，用户可以添加一个整数位 0 来匹配（例如 0.357），所以我认为这并不是一个严重的问题。这个浮点数问题处理起来得靠些诀窍，我们会在第 5 章详细讲解（☞194）。

成功匹配的副作用

Side Effects of a Successful Match

我们再进一步，让这个表达式能够匹配摄氏和华氏温度。我们让用户在温度的末尾加上 C 或者 F 来表示。我们可以在正则表达式的末尾加上「[CF]」来匹配用户的输入，但还需要修改程序的其他部分，以便识别用户输入的温度类型，并进行相应的转换。

在第 1 章，我们看到过某些版本的 *egrep* 支持作为元字符的「\1」、「\2」、「\3」，用来保存前面

的括号内的子表达式实际匹配的文本（☞21）。Perl 和其他许多支持正则表达式的语言都支持这些功能，而且匹配成功之后，在正则表达式**之外**的代码仍然能够引用这些匹配的文本。

我们会在下一章看到各种语言是如何做到这一点的（☞137），但是 Perl 的办法是通过变量 `$1`、`$2`、`$3` 等等，它们指向第一组、第二组、第三组括号内的子表达式实际匹配的文本。这未免有点奇怪，它们**都是**变量，而变量名则是数字。正则表达式匹配成功一次，Perl 就会设置一次。

总结一下，在尝试匹配时，正则表达式中的元字符「\1」指向之前匹配的某些文本，匹配成功之后，在接下来的程序中用`$1` 来引用同样的文本。

为了保持例子的简洁，集中表现新的地方，我先不考虑小数部分，之后再来看它。所以，我们来看`$1`，请比较：

```
$celsius =~ m/^[-+]?[0-9]+[CF]$/
$celsius =~ m/^([-+]?[0-9]+)([CF])$/
```

添加的括号改变了正则表达式的意义吗？为了回答这个问题，我们需要知道，这些括号是否改变了星号或者其他量词的作用对象，或是「|」的意义。答案是，都没有改变，所以这个表达式仍然能够匹配相同的文本。不过，他们确实围住了我们期望匹配字符串中“有价值”文本的子表达式。如图 2-1 所示，`$1` 保存那些数字，而`$2` 保存 `C` 或者 `F`。下一页的图 2-2 是程序的流程图，我们发现，这个图让我们很容易地决定匹配之后应该干什么。

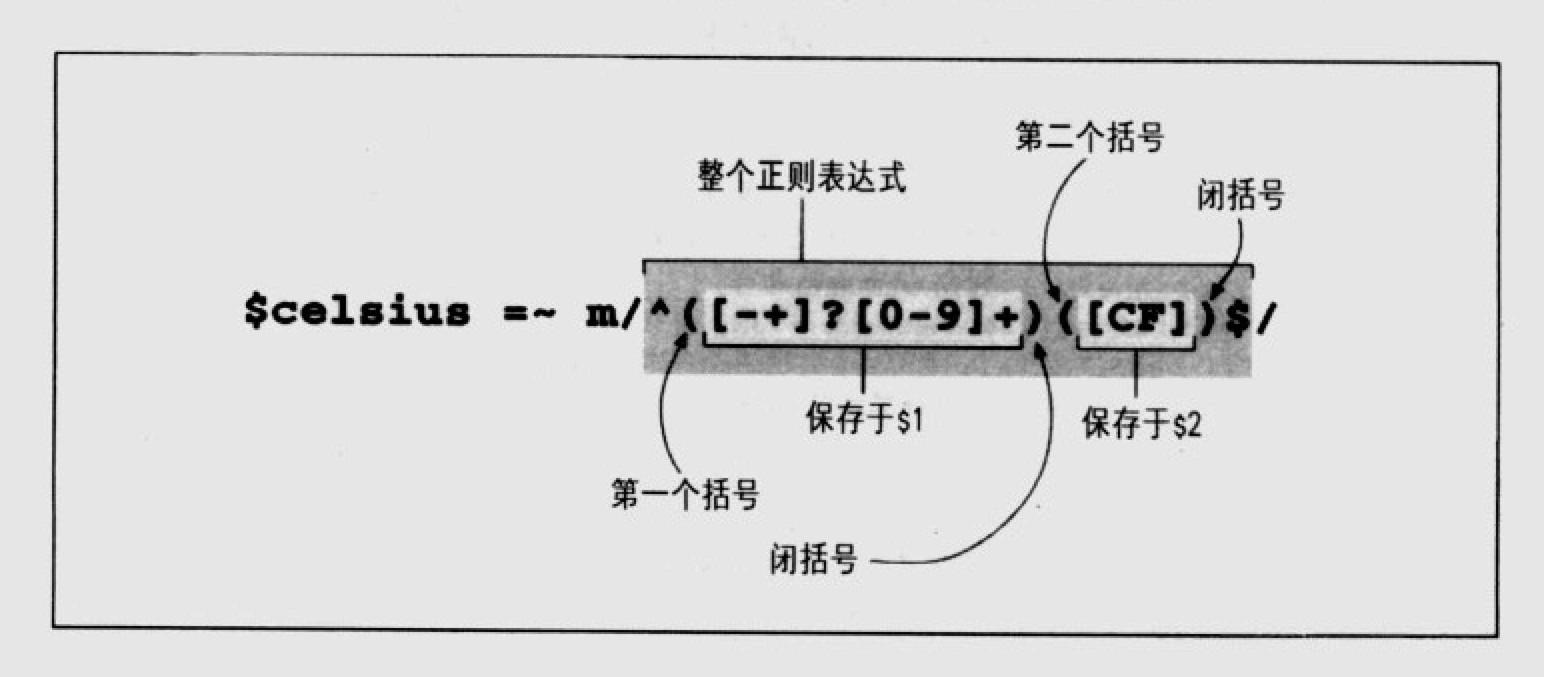


图 2-1：捕获型括号

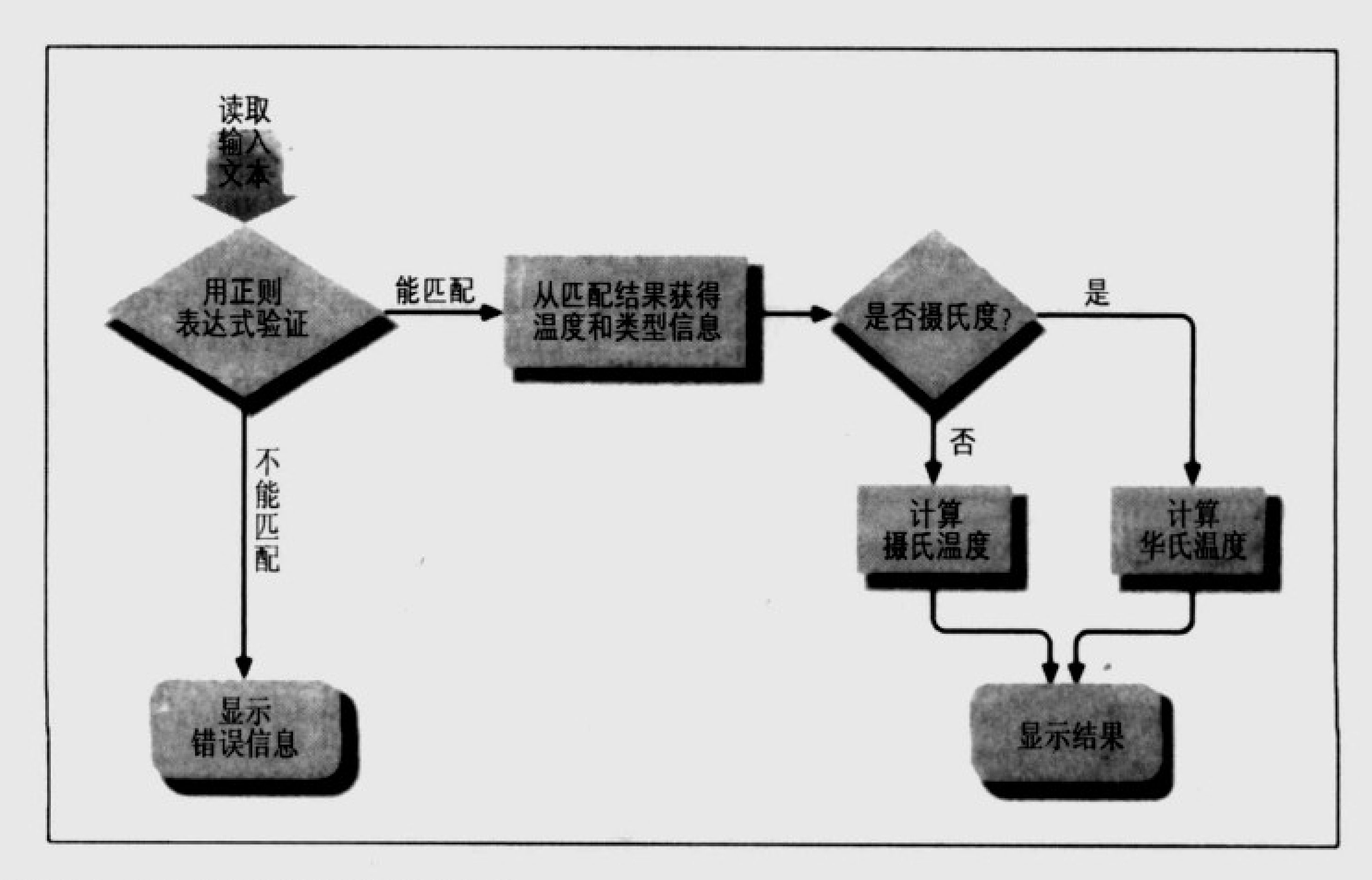


图 2-2：温度转换程序的逻辑流程

示例 2-1：温度转换程序

```
print "Enter a temperature (e.g., 32F, 100C):\n";
$input = <STDIN>; # 接收用户输入的一行文本
chomp($input);     # 去掉文本末尾的换行符

if ($input =~ m/^([-+]?[0-9]+)([CF])$/)
{
    # 如果程序运行到此，则已经匹配。$1 保存数字，$2 保存"C"或者"F"
    $InputNum = $1; #把数据保存到已命名变量中…
    $type     = $2; #…保证程序清晰易懂

    if ($type eq "C") {  # 'eq 测试两个字符串是否相等
        # 输入为摄氏温度，则计算华氏温度
        $celsius = $InputNum;
        $fahrenheit = ($celsius * 9 / 5) + 32;
    } else {
        #如果不是"C"，则必然是"F"，计算摄氏温度
        $fahrenheit = $InputNum;
        $celsius = ($fahrenheit - 32) * 5 / 9;
    }
    #现在得到了两个温度值，显示结果：
    printf "%.2f C is %.2f F\n", $celsius, $fahrenheit;
} else {
    # 如果最开始的正则表达式无法匹配，报警
    print "Expecting a number followed by \"C\" or \"F\",\n";
    print "so I don't understand \"$input\".\n";
}
```

如果上一页的程序名叫 *convert*，我们可以这样使用：

```
% perl -w convert
Enter a temperature (e.g., 32F, 100C):
39F
3.89 C is 39.00 F
% perl -w convert
Enter a temperature (e.g., 32F, 100C):
39C
39.00 C is 102.20 F
% perl -w convert
Enter a temperature (e.g., 32F, 100C):
oops
Expecting a number followed by "C" or "F",
so I don't understand "oops".
```

错综复杂的正则表达式

Intertwined Regular Expressions

在 Perl 之类的高级语言中，正则表达式的使用与其他程序的逻辑是混合在一起的。为了说明这一点，我们对这个程序做三点改进：像之前一样能够接收浮点数，容许 `f` 或者 `c` 是小写，容许数字和字母之间存在空格。这三点全都完成之后，程序就能够接收‘`98.6·f`’的输入。

我们已经知道，添加「`(\.[0-9]*)?`」就能够处理浮点数：

```
if ($input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)([CF])$/)
```

请注意，它添加在第一个括号**内部**，因为我们用第一组括号内的子表达式来捕获温度的值，我们当然希望它能够包含小数部分。不过，增加了这组括号之后，即使它只是用来分组问号限定的对象，也会影响到引用捕获文本的变量。因为这组括号的开括号在整个表达式中排在第二位（从左向右数），所以它匹配的文本存入`$2`（见图 2-3）。

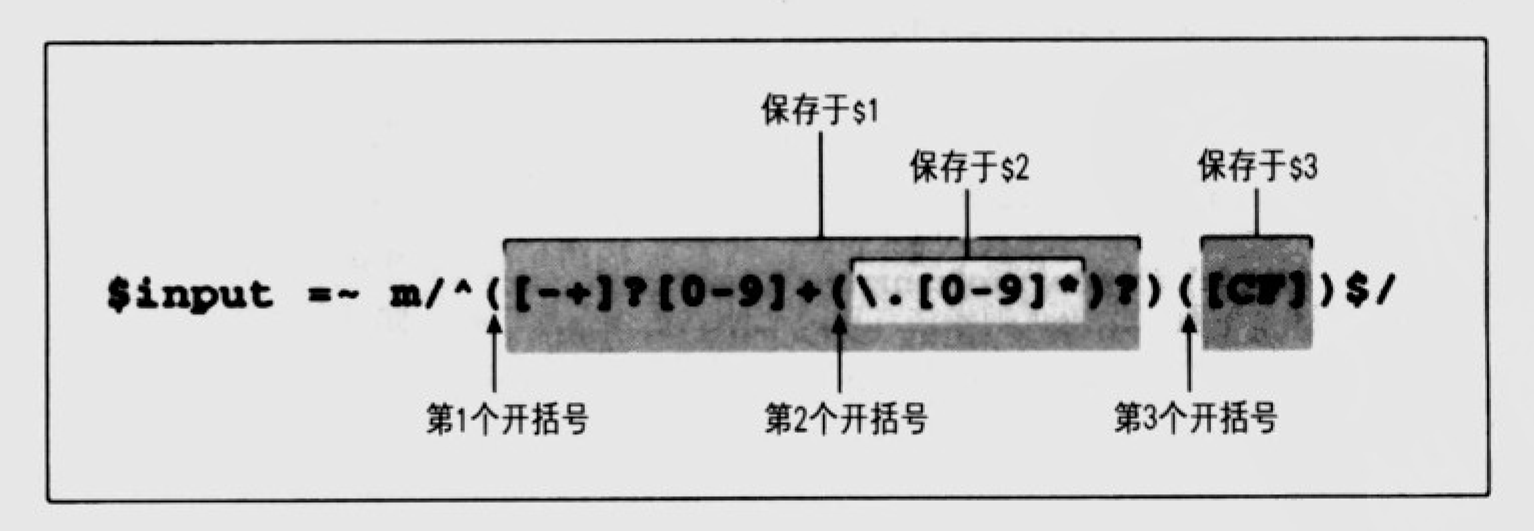


图 2-3：嵌套的括号

图 2-3 说明了括号的嵌套关系。在「[CF]」之前添加一组括号并不会直接影响整个正则表达式的意义，但是会产生间接的影响，因为现在「[CF]」所在的括号排在第 3 位。这也意味着我们需要把$type 的赋值从$2 改为$3（如果不希望这么做，可以参考下一页的补充内容）。

接下来，我们要处理数字和字母之间可能出现的空格。我们知道，正则表达式中的空格字符正好对应匹配文本中的空格字符，所以「·*」能够匹配任意数目的空格（但并不是必须出现空格）：

```
if ($input =~ m/^([-+]?[0-9]+(\.[0-9]*)?) *([CF])$/)
```

但这样还不够灵活，而我们希望的是开发一个能够实际应用的程序，所以必须容许其他的**空白字符**（whitespace）。例如常见的制表符（tabs）。但是「⇥*」并不能匹配空格，所以我们需要一个字符组来匹配两者「[·⇥]*」。

请把上面这个子表达式与「(·*|⇥*)」进行对比，你能发现这其中的巨大差异吗？◆请翻到下一页查看答案。

本书中空格字符和制表符都很常见，因为我使用·和⇥来表示它们。不幸的是，在屏幕上却不是如此。如果你见到[ ]*，在没有实际测试过以前，只能猜测这是空格符还是制表符。为了方便使用，Perl 提供了「\t」这个元字符。它能够匹配制表符——相比真正的制表符，它的好处就在于看得更清楚，所以我会在正则表达式中采用这个元字符。于是「[·⇥]*」变成「[·\t]*」。

Perl 还提供了一些简便的元字符，例如「\n」(表示换行符)，「\f」(ASCII 的进纸符 formfeed)，和「\b」(退格符)。不过，确切地说，「\b」在某些情况下是退格符，有些情况下又表示单词分界符。它怎么能身兼数职呢？下一节我们会看到。

一点题外话——数量丰富的元字符

在前面的例子里我们见到了\n，但是\n 都出现在字符串而不是正则表达式中。就像多数语言一样，Perl 的**字符串**也有自己的元字符，它们完全不同于**正则表达式**元字符。新程序员常犯的错误就是混淆了这两个概念（VB.NET 是个例外，因为其中字符串的元字符少得可怜）。字符串的元字符中有一些跟正则表达式中对应的元字符一模一样。你可以在字符串中用\t 加入制表符，也可以在正则表达式中用元字符「\t」来匹配制表符。

非捕获型括号「(?:…)」

在图 2-3 中，我们用括号包围「(\.[0-9]*)?」来正确分组，所以我们能够用一个问号正确地作用于整个「\.[0-9]*」，把它们作为可选项部分。这样的副作用就是这个括号内的子表达式捕获的文本保存到$2 中，而我们并不会使用$2。如果有这样一种括号，它只能用于分组，而不会影响文本的捕获和变量的保存，问题就好办多了。

Perl 以及近期出现的其他正则表达式流派提供了这个功能。「(…)」用来分组和捕获，而「(?:…)」表示**只分组不捕获**。使用这个表示法，"开括号"是 3 个字母构成的序列(?:，这看起来很古怪。这里的 '?' 和表示"可选项"的「?」元字符没有任何联系（可以翻到第 90 页了解这个古怪的表示法的来历）。

所以，整个表达式变成：

```
if ($input =~ m/^([-+]?[0-9]+ (?:\.[0-9]*)?)([CF])$/)
```

现在，即使「[CF]」两端的括号的确是排在第三位，它匹配的文本也会保存到$2 中，因为「(?:…)」不会影响捕获计数。

这样做的好处有两点。第一是避免了不必要的捕获操作，提高了匹配效率（我们会在第 6 章详细讨论效率问题）。另一个好处是，总的来说，根据情况选择合适的括号能够让程序更清晰，看代码的人不会被括号的具体细节所困扰。

另一方面，「(?:…)」表示法确实不够美观，或者说它会增加整个表达式的阅读难度。它带来的好处能够抵消这些问题吗？我个人喜欢根据需要选择合适的括号，但是在本例中，或许不值得这样麻烦。如果匹配只需要进行一次（而不需要在循环中多次匹配），效率并不重要。

在本章中，我全部采用「(…)」，即使不需要捕获文本时也是如此，因为这样看起来更清楚。

这种相似性无疑方便了使用，但是我必须强调区分这两种元字符的重要性。对于\t 这样简单的情况来说或许并不重要，但对于我们将要看到的各种不同的语言和工具来说，知道在什么情况下应该使用什么元字符是极其重要的。

测验答案

◆ 44 页问题的答案

「[·⇥]*」和「(·*|⇥*)」的异同

「(·*|⇥*)」容许「·*」或「⇥*」的匹配，它能够匹配若干空格符（也可以没有）以及若干制表符（也可以没有），不过并不容许制表符和空格符的**混合体**。

相反，「[·⇥]*」能够匹配任意多个「·⇥」。对于字符串‘⇥··’，它可以匹配 3 次，第一次是制表符，后两次是空格符。

在逻辑上，「[·⇥]*」与「(·|⇥)*」是等价的，尽管因为第 4 章将会解释其原因，字符组的效率通常还是会高一点。

我们已经见过不同的字符组之间的冲突。在第 1 章，使用 *egrep* 时，我们把正则表达式包含在单引号中。整个 *egrep* 命令行写在 command-shell 提示符，shell 能够认出它自己的元字符。例如，对 shell 来说，空格符就是一个元字符，它用来分隔命令和参数，或者参数与参数。在许多 shell 中，单引号是元字符，单引号内的字符串中的字符不需要被当作元字符处理（DOS 使用双引号）。

在 shell 中使用引号容许我们在正则表达式中使用空格。否则，shell 会把空格认作参数之间的分隔符，而不是把整个正则表达式传递给 *egrep*。许多 shell 能够识别的元字符包括 $、*、? 之类——我们在正则表达式中也会用到这些元字符。

目前，所有关于 shell 的元字符和 Perl 字符串的元字符的讨论都还与正则表达式本身没有任何关联，但它们会影响到现实环境中正则表达式的使用。随着阅读的深入，我们会见到许多（有时候还很复杂）情况，我们需要同时在不同层级上使用元字符交互（multiple levels of simultaneously interacting metacharacters）。

那么「\b」的情况呢？这是一个正则表达式的问题：在 Perl 的正则表达式中，「\b」通常是匹配一个单词分界符的，但是在字符组中，它匹配一个退格符。单词分界符作为字符组的一部分则没有任何意义，所以 Perl 完全可以用它来匹配其他的字符。第 1 章曾提醒我们，字符组“子语言”的规范不同于正则表达式主体，这条规则也适用于 Perl（包括任何其他流派的正则表达式）。

用\s 匹配所有"空白"

讨论空白的问题时，我们最后使用的是「[·\t]*」。这样做没问题，但许多流派的正则表达式提供了一种方便的办法：「\s」。「\s」看起来类似「\t」，「\t」代表制表符，而「\s」则能表示所有表示"空白字符（whitespace character)"的字符组，其中包括空格符、制表符、换行符和回车符。在我们的例子中，换行符和回车符并不需要特别考虑，但是「\s*」显然比「[·\t]*」要简洁。而且不久你就会习惯这种表示法，在复杂的表达式中，「\s*」更加易于理解。

现在我们的程序变成：

```
$input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/
```

最后，我们还必须能够处理表示温度制式的小写字母。简单的办法是直接把小写字母添加到字符组中，「[CFcf]」。不过，我更愿意使用另一种办法：

```
$input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i
```

添加的这个 i 称作"**修饰符**（modifier)"，把它放在 m/…/结构之后，告诉 Perl 进行不区分大小写的匹配。修饰符其实不是正则表达式的一部分，而是 m/…/结构的一部分，这个结构告诉 Perl 使用者的意图（应用一个正则表达式)，以及采用的正则表达式（在斜线之间的部分)。我们曾看到过这种功能，即 *egrep* 的-i 参数（☞15)。

时时刻刻说"i 修饰符（the i modifier)"有点麻烦，所以我们通常说"/i"，即使真正的写法并不是"/i"。/i 只是在 Perl 中指定修饰符的办法之一——在下一章，我们会看到其他的办法，以及其他语言实现此功能的写法。在本章后面的部分，我们还会看到其他的修饰符，例如/g（表示"全局匹配（global match)"）以及/x（表示"宽松排列的表达式（free-form expressions)"）。

现在，我们已经做了不少修改了，来看看新的程序：

```
% perl -w convert
Enter a temperature (e.g., 32F, 100C):
32 f
0.00 C is 32.00 F
% perl -w convert
Enter a temperature (e.g., 32F, 100C):
50 c
10.00 C is 50.00 F
```

哎呀！你是否注意到了，第二次运行时我们输入的是摄氏 50 度，结果被认成了华氏 50 度？看看程序的逻辑，你找出问题了吗？

再来看程序的片段：

```
if ($input =~ m/^[-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i)
{
  ......

  $type = $3; #把数据保存到以命名的变量，让程序更易懂

  if ($type eq "C") { # 'eq' 测试两个字符串是否相等
     ......

  } else {
     ......
```

虽然我们的正则表达式能够接受小写的 f，程序的其他部分却没有相应的修改。在这个程序里，只有$type 是 'C' 的时候，才作为摄氏度处理。因为程序同样可以接受小写的 c，我们需要修改$type 的判断：

```
if ($type eq "C" or $type eq "c") {
```

实际上，因为本书是关于正则表达式的，我或许这样做：

```
if ($type =~ m/c/i) {
```

现在，大小写的情况都能应付了。最终的程序如下所示。这个例子告诉我们，正则表达式的使用方式，可能会影响到程序的其他部分。

示例 2-2：温度转换程序——最终版本

```
print "Enter a temperature (e.g., 32F, 100C):\n";
$input = <STDIN>;      # 接收用户输入的一行文本
chomp($input);         # 去掉$input 末尾的换行
if ($input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i)
{
   # 如果运行到此，则已经匹配，$1 保存数值，$3 保存 C 或者 F
   $InputNum = $1;    # 把数据保存到已命名的变量
   $type     = $3;    # ......保证程序的可读性
   if ($type =~ m/c/i) {     # Is it "c" or "C"?
      #输入的是摄氏温度，计算华氏温度
      $celsius = $InputNum;
      $fahrenheit = ($celsius * 9 / 5) + 32;
   } else {
      # 如果不是"C"，则必定是"F"，所以计算摄氏温度
      $fahrenheit = $InputNum;
      $celsius = ($fahrenheit - 32) * 5 / 9;
   }
   # 现在两个值都有了，输出结果：
   printf "%.2f C is %.2f F\n", $celsius, $fahrenheit;
} else {
   # 开始的表达式无法匹配，发出警报
   print "Expecting a number followed by \"C\" or \"F\",\n";
   print "so I don't understand \"$input\".\n";
}
```

暂停片刻

Intermission

尽管本章的大部分篇幅是关于熟悉 Perl 的，但也遇到了许多新的关于正则表达式的知识。

1. 许多工具都有自己的正则表达式流派。Perl 和 *egrep* 可能属于同一个流派，但是 Perl 的正则表达式中的元字符更多。许多其他的语言，类似 Java、Python、.NET 和 TCL，它们的流派类似 Perl。

2. Perl 用**`$variable =~ m/`*regex*`/`**来判断一个正则表达式是否能匹配某个字符串。`m` 表示“**匹配**（match）”，而斜线用来标注正则表达式的边界（它们本身不属于正则表达式）。整个测试语句作为一个单元，返回 true 或者 false 值。

3. 元字符——具有特殊意义的字符——的定义在正则表达式中并不是统一的。之前在关于 shell 和双引号引用的字符串的例子中我们讲过，元字符的含义取决于具体的情况。了解具体情况（shell、正则表达式、字符串），其中的元字符及其作用，对学习和使用 Perl、PHP、Java、Tcl、GNU Emacs、awk、Python 或其他高级语言是非常重要的（当然，在正则表达式内部，字符组有自己的“子语言”，其中的元字符是不同的）。

4. Perl 和其他流派的正则表达式提供了许多有用的简记法（shorthands）：

```
\t  制表符
\n  换行符
\r  回车符
\s  任何“空白”字符（例如空格符、制表符、进纸符等）
\S  除「\s」之外的任何字符
\w  「[a-zA-Z0-9]」（在「\w+」中很有用，可以用来匹配一个单词）
\W  除「\w」之外的任何字符，也就是「[^a-zA-Z0-9]」
\d  「[0-9]」，即数字
\D  除「\d」外的任何字符，即「[^0-9]」
```

5. `/i` 修饰符表示此测试不区分大小写。尽管写法是“`/i`”，其实“`i`”只是跟在表示结尾的斜线之后。

6. 「`(?:…)`」这个麻烦的写法可以用来分组文本，但并不捕获。

7. 匹配成功之后，Perl 可以用`$1`、`$2`、`$3` 之类的变量来保存相对应的「`(…)`」括号内的子表达式匹配的文本。使用这些变量，我们能够用正则表达式从字符串中提取信息（其他的语言所使用的方式有所不同，我们会在下一章看到例子）。

子表达式的编号按照开括号的出现先后排序，从 1 开始。子表达式可以嵌套，例如「(Washington(•DC)?)」。如果只是希望分组，也可以使用「(…)」，但副作用是，它们捕获的文本仍然会保存到特殊的变量中。

使用正则表达式修改文本

Modifying Text with Regular Expressions

到现在，我们遇到的例子都只是从字符串中“提取”信息。现在我们来看 Perl 和其他许多语言提供的一个正则表达式特性：**替换**（substitution，也可以叫“**查找和替换**（search and replace)”）。

我们已经看到，**$var =~ m/*regex*/**尝试用正则表达式来匹配保存在变量中的文本，并返回表示能否匹配的布尔值。与之类似的结构**$var =~ s/*regex*/*replacement*/**则更进一步：如果正则表达式能够匹配`$var`中的某段文本，则将这段匹配的文本替换为 *replacement*。其中 regex 与之前 `m/…/`的用法一样，而 replacement（位于第二个和第三个斜线之间）则是作为双引号内的字符串。这就是说，在其中可以使用变量——例如`$1`、`$2`——来引用之前匹配的具体文本。

所以，使用**$var =~ s/…/…/**可以改变`$var`中的文本（如果没有找到匹配的文本，也就不会有替换发生）。例如，如果`$var`包括 `Jeff•Friedl`，运行：

```
$var =~ s/Jeff/Jeffrey/;
```

`$var` 的值就变成 `Jeffery•Friedl`。如果再运行一次，就得到 `Jeffreyrey•Friedl`。要避免这种情况，也许我们需要添加表示单词分界的元字符。在第 1 章我们提到过，某些版本的 *egrep* 支持「\<」和「\>」作为“单词起始”和“单词结束”的元字符。Perl 提供了统一的元字符「\b」来代表这两者：

```
$var =~ s/\bJeff\b/Jeffrey/;
```

这里有个小测验：与 `m/…/`一样，`s/…/…/`也可以使用修饰符，例如第 47 页介绍的`/i`（将这个修饰符放在 replacement 之后）。那么，这个表达式：

```
$var =~ s/\bJeff\b/Jeff/i;
```

的功能是什么呢？◆请翻到下页查看答案。

例子：公函生成程序

Example: Form Letter

下面这个有趣的例子展示了文本替换的用途。设想有一个公函系统，它包含很多公函模板，其中有一些标记，对每一封具体的公函来说，标记部分的值都有所不同。

这里有一个例子：

```
Dear =FIRST=,
You have been chosen to win a brand new =TRINKET=! Free!
Could you use another =TRINKET= in the =FAMILY= household?
Yes =SUCKER=, I bet you could! Just respond by……
```

对特定的接收人，变量的值分别为：

```
$given = "Tom";
$family = "Cruise";
$wunderprize = "100% genuine faux diamond";
```

准备好之后，就可以用下面的语句“填写模板”：

```
$letter =~ s/=FIRST=/$given/g;
$letter =~ s/=FAMILY=/$family/g;
$letter =~ s/=SUCKER=/$given $family/g;
$letter =~ s/=TRINKET=/fabulous $wunderprize/g;
```

其中的每个正则表达式首先搜索简单标记，找到之后用指定的文本替换它。用于替换的文本其实是 Perl 中的字符串，所以它们能够引用变量，就像上面的程序那样。例如，`s/=TRINKET=/fabulous $wunderprize/g`中下画线部分在程序运行时的值就是“`fabulous $wunderprize`”。如果只需要生成一份公函，完全可以不用变量替换，直接照需要的样子生成就是。但是，使用变量替换能够实现自动化的操作，例如可以从一个清单读入信息。

我们还没介绍过`/g`“全局替换”（global replacement）的修饰符。它告诉`s/…/…/`在第一次替换完成之后继续搜索更多的匹配文本，进行更多的替换。如果需要检查的字符串包含多行需要替换的文本，每条替换规则都对所有行生效，我们就必须使用`/g`。

结果是可以预见的，不过相当有趣：

```
Dear Tom,
You have been chosen to win a brand new fabulous 100% genuine faux diamond!
Free! Could you use another fabulous 100% genuine faux diamond in the Cruise
household? Yes Tom Cruise, I bet you could! Just respond by.....
```

举例：修整股票价格

Example: Prettifying a Stock Price

另一个例子是，我在使用 Perl 编写的股票价格软件时遇到的问题。我得到的价格看起来是这样“`9.0500000037272`”。这里的价格显然应该是 `9.05`，但是因为计算机内部表示浮点的原理，Perl 有时会以没什么用的格式输出这样的结果。我们可以像温度转换例子中的那样

测验答案

◆ 50 页的测验的答案

$var =~ s/\bJeff\b/Jeff/i 实现了什么功能？

这个问题可能让你困惑。如果前面的正则表达式用「\bJEFF\b」或者「\bjeff\b」或者是「\bjEfF\b」可能看得更清楚。因为/i 的存在，搜索“Jeff”这个词是不区分大小写的。而所有匹配的字符串都会被替换为‘Jeff’，第一个字母是大写，其他为小写（/i 对 replacement 的文本没有影响，不过第 7 章讨论的某些修饰符不是如此）。

结果就是，“jeff”这个单词，无论大小写的情况如何，都会被替换为“Jeff”。

用 printf 来保证只输出两位小数，但是此处并不适用。当时，股价仍然是以分数的形式给出的，如果某个价格以 1/8 结尾，则应该输出 3 位小数（“.125”），而不是两位。

我把自己的要求归结为：通常是保留小数点后两位数字，如果第三位不为零，也需要保留，去掉其他的数字。结果就是 12.3750000000392 或者 12.375 会被修正为“12.375”，而 37.500 被修正为“37.50”。这就是我要的结果。

那么，我们该如何做呢？$price 变量包含了需要修正字符串，让我们用这个表达式：

```
$price =~ s/(\.\d\d[1-9]?)\d*/$1/
```

（提示：49 页介绍了「\d」这个元字符，它用来匹配一个数字字符。）

最开始的「\.」匹配小数点。接下来的「\d\d」匹配开头的两位数字。「[1-9]?」匹配可能跟在后面的非零数字。到这里，任何匹配的文本都是我们希望保留的，所以我们用括号把它保存到$1 中。然后将$1 放入 replacement 字符串中。如果能够匹配的文本就是$1，我们就用$1 替换$1——这样做没什么意义。但如果在$1 的括号之外还有能够匹配的字符，因为它们没有出现在 replacement 字符串中，所以会被删除。也就是说，“被删除的”文本是其他多余的数字，也就是正则表达式末尾「\d*」匹配的字符。

请记住这个例子，在第 4 章我们会学习匹配过程背后的重要原理，那时候还会遇到这个例子。研究它可以学到非常有价值的知识。

自动的编辑操作

Automated Editing

写作本章时，我遇到了另一个简单但真实存在的例子。当时我需要登录到太平洋对岸的一台机器上，但是网速非常慢。按下回车得等一分多钟才能见到反应，而我只需要对某个文件进行一些小的改动，运行一个重要的程序。实际上，我要做的只是将出现的所有 `sysread` 改为 `read`。改动的次数并不多，但因为网络太慢，使用全屏编辑器显然是不可能的。

下面是我的办法：

```
% perl -p -i -e 's/sysread/read/g' file
```

这条命令中的 Perl 程序是 **s/sysread/read/g**（是的，这就是一个完整的 Perl 程序——参数 `-e` 表示整个程序接在命令的后面）。参数 `-p` 表示对目标文件的每一行进行查找和替换，而 `-i` 表示将替换的结果写回到文件。

请注意，这里没有明确写出查找和替换的目标字符串（就是说，没有 **$var =~**…），因为 `-p` 参数就表示对目标文件的每行文本应用这段程序。同样，因为我用了 `/g` 这个修饰符，就可以保证在一行文本中可以进行多次替换。

尽管在这里我只是对一个文件进行操作，但也很容易在命令行中列出多个文件，而 Perl 会把替换命令应用到每个文件的每一行文字。这样，只需要一条简单的命令，我就能够编辑大量的文件。这样简单的编辑方式是 Perl 独有的，但这个例子告诉我们，即使执行的是简单的任务，作为脚本语言一部分的正则表达式的功能仍然非常强大。

处理邮件的小工具

A Small Mail Utility

来看另一个小工具的例子。一个文件中保存着 E-mail 信息，我们需要生成一个用于回复的文件。在准备过程中，我们需要引用原始的信息，这样就能很容易地把回复插入各个部分。在生成回复邮件的 header 时，我们还需要删除原始信息邮件的 header 中不需要的行。

下一页的补充内容是一个邮件文件的范本。header 包含了我们关心的字段：日期、主题等——但也包括了我们不关注的字段，这些字段需要删除。如果我们的脚本程序叫做 *mkreply*，而原始的信息保留在 *king.in* 中，我们会用下面的命令来生成回复模板：

```
% perl -w mkreply king.in > king.out
```

（`-w` 它用来打开 Perl 的额外警告功能，☞38）

E-mail Message 范本

```
From elvis Thu Feb 29 11:15 2007
Received: from elvis@localhost by tabloid.org (8.11.3) id KA8CMY
Received: from tabloid.org by gateway.net (8.12.5/2) id N8XBK
To: jfriedl@regex.info (Jeffrey Friedl)
From: elvis@tabloid.org (The King)
Date: Thu, Feb 29 2007 11:15
Message-Id: <2007022939939.KA8CMY@tabloid.org>
Subject: Be seein' ya around
Reply-To: elvis@hh.tabloid.org
X-Mailer: Madam Zelda's Psychic Orb [version 3.7 PL92]

Sorry I haven't been around lately. A few years back I checked into
that ole heartbreak hotel in the sky, ifyaknowwhatImean.
The Duke says "hi".
        Elvis
```

我们希望程序的输出结果 *king.out* 包括下面的内容：

```
To: elvis@hh.tabloid.org (The King)
From: jfriedl@regex.info (Jeffrey Friedl)
Subject: Re: Be seein' ya around

On Thu, Feb 29 2007 11:15 The King wrote:
|> Sorry I haven't been around lately. A few years back I checked
|> into that ole heartbreak hotel in the sky, ifyaknowwhatImean.
|> The Duke says "hi".
|>        Elvis
```

现在我们来分析。为了生成新的 header，我们需要知道目标地址（即本例中的 `elvis@hh.tabloid.org`，来自原始信息中的 `Reply-To` 字段），收件人的姓名（`The King`），我们自己的地址和姓名，以及主题。另外，为了生成邮件正文的导入部分（introductory line），我们还需要知道原始邮件的日期。

这些工作可以分为下面 3 步：

1. 从原始邮件的 header 中提取信息；
2. 生成回复邮件 header；
3. 打印原始邮件信息，行首用‘`|>·`’缩进。

这样考虑有点超前了——在没有决定程序如何读入数据之前，就关心起如何处理数据了。幸运的是，Perl 提供了神奇的“<>”操作符。在应用到变量`$variable`时，使用“`$variable = <>`”，这个有趣的结构能够每次读入一行数据。输入的数据来自命令行中 Perl 脚本之后列出的文件名（例如上面例子中的 *king.in*）。

请不要混淆操作符<>与 Shell 的重定向符号“> *filename*”或者是 Perl 的大于/小于号。Perl 中的<>相当于其他语言中的 `getline()` 函数。

读入所有输入数据之后，<>很方便地返回未定义的值（作为布尔值处理），所以整个文件可以这样处理：

```
while ($line = <>) {
…处理$line…
}
```

我们会用类似的办法来处理邮件，但是邮件本身的性质决定了我们必须对邮件 header 特殊处理。第一个空行之前的信息是 header，之后的则是正文部分。为了只读入 header，我们可以使用下面这段代码。

```
# 处理 header
while ($line = <>) {
   if ($line =~ m/^\s*$/) {
       last; # 停止 while 循环内的处理，跳出循环
   }
   ...处理 header 信息...
}
...处理邮件内的其他信息...
   ……
```

我们用「^\s*$」来检查表示邮件 header 结束的空行。这个正则表达式检查的是，当前的文本行是否有一个行开头（其实每一行都有，由脱字符匹配），然后跟着任意数目的空白字符（尽管我们并不期望有任何空白字符），然后字符串结束（注 3）。关键词 last 会跳出 while 循环，停止处理 header。

所以，在循环内部，在空行检测之后，我们能够按照自己的想法来处理 header 的每一行。在本例中，我们希望提取信息，例如邮件的主题和时间。

要提取主题，我们可以使用一个常见的技巧：

```
if ($line =~ m/^Subject: (.*)/i) {
    $subject = $1;
}
```

这段代码尝试匹配一个以‘Subject:·’开头，但不区分 Subject 大小写的字符串。如果能够匹配，后面的「.*」匹配这一行的其他部分。因为「.*」在括号中，所以之后我们能用$1 来访问邮件的主题。在这个例子中，我们希望把它保存到变量$subject 中。当然，如果正则表达式无法匹配这个字符串（大多数情况下都不能），结果就是 if 语句返回结果为 false，$subject 变量没有变化。

注 3：我在这里用的是“字符串”（string）而不是“行”（line），因为虽然对本例来说这不是一个问题，但正则表达式需要处理的可能是一个包含多行文本的字符串。通常，脱字符和美元符只匹配整个字符串的开头和结尾（在本章后面我们会遇到一个相反的例子）。无论如何，此处这种区别并不重要，因为根据我们的程序，我们知道$line 的内容不会多于一行。

关于「.*」的警告

「.*」通常用来表示"一组任何字符"（a bunch of anything），因为点号可以匹配任何字符（在某些工具中，不包括换行符），而星号表示可以为任意数目，但并非必须。「.*」可能很有用。

不过，如果把「.*」作为整个正则表达式的一部分，而用户又不真正了解其中的原理，就可能陷入某些隐藏的"陷阱"。我们已经看到过一个例子（☞26），并会在第 4 章深入讨论这个话题的时候见到更多的例子（☞164）。

我们可以用同样的办法来处理 Date 和 Reply-To 字段：

```
if ($line =~ m/^Date: (.*)/i) {
  $date = $1;
}
if ($line =~ m/^Reply-To: (.*)/i) {
  $reply_address = $1;
}
```

From:所在的行稍微麻烦一点。首先，我们需要找到以 'From:␣' 开头的行，而不是以 'From␣' 开头的第一行。我们需要的是：

```
From: elvis@tabloid.org (The King)
```

它包含了邮件的发送地址，发送者的姓名在括号内，我们要提取的是姓名（译注 1）。

我们用「^From:·(\S+)」来提取发送地址。你可能猜到了，「\S」匹配的是所有的非空白字符（☞49），所以「\S+」匹配第一个空白之前的文本（或者目标文本末尾之前的所有字符）。在本例中，就是邮件的发送地址。匹配之后，我们希望匹配括号内的文字。显然，此处也需要匹配括号本身。我们用「\(」和「\)」来匹配，转义之后的括号不再具有特殊的含义。在括号内，我们希望匹配任何字符——除了括号之外的任何字符，所以采用「[^()]*」。记住，字符组的元字符不同于正则表达式的"普通"元字符，在字符组内部，括号不再具有特殊含义，因此也不需要转义。

综合起来，我们得到：

「^From:·(\S+)·\(([^()]*)\)」

其中的括号有点多，初看起来不太好懂，图 2-4 解释得更清楚：

译注 1：一般情况下，邮件应当回复到"回复地址"，即 Reply-To 字段的内容，若此字段不存在，则回复到发送的邮箱，故此处说需要提取的只是发送者的姓名。

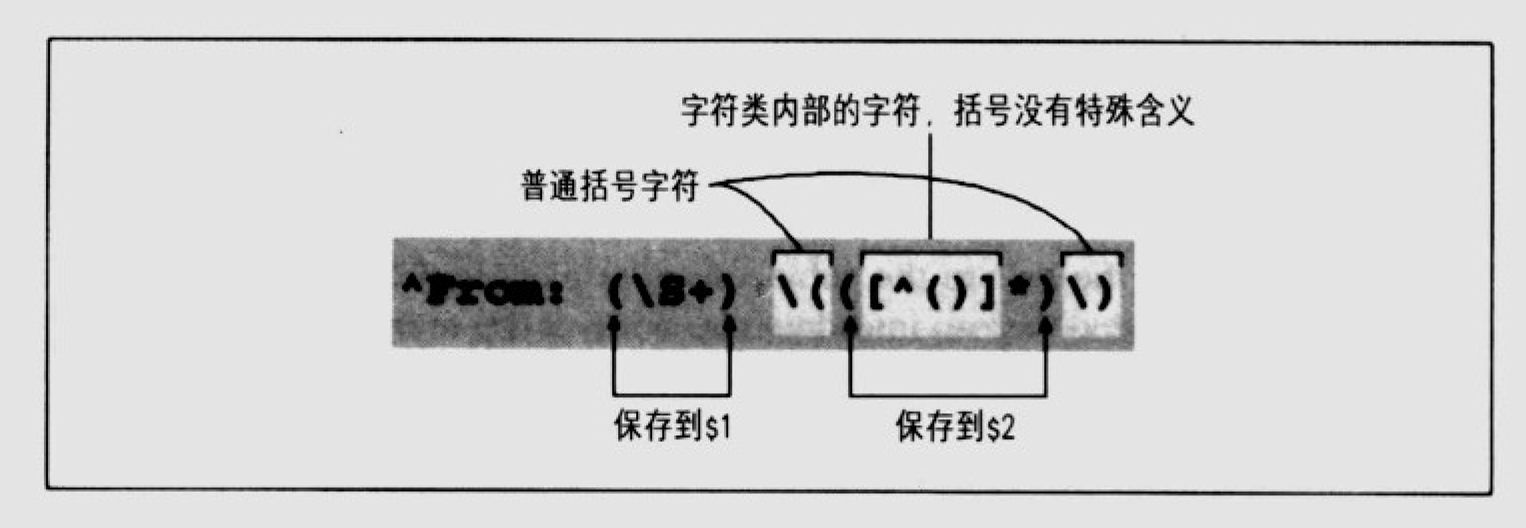


图 2-4：嵌套的括号，$1 和$2

如果图 2-4 的正则表达式能够匹配，我们可以通过`$2` 得到发送者的姓名，从`$1` 得到可能的回复地址。

```
if ($line =~ m/^From: (\s+) \(([^()]*)\)/i) {
    $reply_address = $1;
    $from_name = $2;
}
```

并非所有的 E-mail 信息都包含 `Reply-To` 字段，所以我们把`$1` 暂定为回复地址。如果之后出现了`$Reply-To` 字段，我们会重设`$reply_address`。综合起来就得到：

```
while ($line = < >)
{
    if ($line =~ m/^\s*$/ ) { # 如果存在空行
        last; # 就立即结束 while 循环
}
    if ($line =~ m/^Subject: (.*)/i) {
        $subject = $1;
}
    if ($line =~ m/^Date: (.*)/i) {
        $date = $1;
}
    if ($line =~ m/^Reply-To: (\s+)/i) {
        $reply_address = $1;
}
    if ($line =~ m/^From: (\s+) \(([^()]*)\)/i) {
        $reply_address = $1;
        $from_name = $2;
    }
}
```

这段程序检查 header 的每一行，如果某个正则表达式能够匹配，则设置相应的变量。header 的许多行无法由这些正则表达式匹配，所以会被忽略。

while 循环结束之后，我们就能够生成回复邮件的 header 了（注 4）：

```
print "To: $reply_address ($from_name)\n";
print "From: jfriedl\@regex.info (Jeffrey Friedl)\n";
print "Subject: Re: $subject\n";
print "\n" ; # blank line to separate the header from message body.
```

请注意，我们在主题之前加上了 Re:，表示这是一封回复邮件。最后，在 header 之后，我们列出原始邮件的内容：

```
print "On $date $from_name wrote:\n";
```

对于其他的输入信息（也就是原始邮件的正文部分），我们在每一行之前添加‘|>·’提示符：

```
while ($line = < >) {
   print "|> $line";
}
```

有意思的是，这段程序也可以用另一种方法，使用正则表达式来加入引用提示符：

```
$line =~ s/^/|> /;
print $line;
```

这条替换命令寻找「^」，在每个字符串的起始位置匹配。这条替换命令把字符串开头那个“不存在的字符”“替换”为‘|>·’，其实并没有替换任何字符，只是在字符串的开头加入‘|>·’。在本例中这样做有点滥用的嫌疑了，但是我们将在本章中看到类似（但更有用）的例子。

真实世界的问题，真实世界的解法

既然摆出了一个真实世界的例子，就应该指出这个解法在真实世界中的缺憾。我已经说过，这些例子的目的在于展示正则表达式的使用方法，而 Perl 程序不过是展示的手段。我使用的 Perl 程序并不一定使用了最有效或者最好的解法，但是，我希望它能说明正则表达式的用法。

同样，真实世界的邮件信息比这个简单问题中的邮件信息复杂很多。From:这一行就可能有许多种格式，而我们的程序只能处理一种。如果真正的 From:这一行无法匹配我们的模式，则$from_name 变量就不会设置，使用时保持在未定义的状态（也就是“没有值”的值的一种）。理想的解决办法是修改这个正则表达式，让它能够处理各种不同的邮件地址/姓名格式，

注 4：在 Perl 的正则表达式和双引号内的字符串中，大多数‘@’都必须转义（☞77）。

不过，作为第一步，在检查原始邮件之后（生成回复模板之前），我们可以这样：

```
if (    not defined($reply_address)
     or not defined($from_name)
     or not defined($subject)
     or not defined($date) )
{
     die "couldn't glean the required information!";
}
```

Perl 的 defined 函数检查一个变量是否设置了值，而 die 函数用来发出错误信息，退出程序。

另一点需要考虑的是，程序假设 From:这一行出现在 Reply-To:之前。如果 From:出现在之后，就会覆盖从 Reply-To 取得的$reply_address。

"真正的"真实世界

发送电子邮件的程序有许多类，每类程序对标准的理解都不一样，所以处理电子邮件并不是件简单的事情。我曾经想用 Pascal 程序来处理电子邮件，但我发现，如果没有正则表达式，处理起来极其困难，困难到我决定先用 Pascal 写一个类似 Perl 的正则表达式包，再来做其他事情。进入没有正则表达式的世界之后才发现，自己已经习惯正则表达式的功能和便捷了，而我显然不希望在没有正则表达式的世界呆太久。

用环视功能为数值添加逗号

Adding Commas to a Number with Lookaround

大的数值，如果在其间加入逗号，会更容易看懂。下面的程序：

```
print "The US population  is $pop\n";
```

可能输出"The US population is 298444215"，但对大多数说英语的人来说，"298,444,215"看起来更加自然。用正则表达式该如何做呢？

动脑子想想这个问题，我们应该从这个数的右边开始，每次数 3 位数字，如果左边还有数字的话，就加入一个逗号。如果我们能把这种思路直接用到正则表达式中当然很好，可惜正则表达式一般都是从左向右工作的。不过梳理一下思路就会发现，逗号应该加在"左边有数字，右边数字的个数正好是 3 的倍数的位置"，这样，使用一组相对较新的正则表达式特性——它们统称为"环视（lookaround）"——轻松地解决这个问题。

环视结构不匹配任何字符，只匹配文本中的**特定位置**，这一点与单词分界符「\b」、锚点「^」

和「$」相似。但是，环视比它们更加通用。

一种类型的环视叫**“顺序环视（lookahead）”**，作为表达式的一部分，顺序环视顺序（从左至右）查看文本，尝试匹配子表达式，如果能够匹配，就返回匹配成功信息。肯定型顺序环视（positive lookahread）用特殊的序列「(?=…)」来表示，例如「(?=\d)」，它表示如果当前位置右边的字符是数字则匹配成功。另一种环视称为逆序环视，它逆序（从右向左）查看文本。它用特殊的序列「(?<=…)」表示，例如「(?<=\d)」，如果当前位置的左边有一位数字，则匹配成功（也就是说，紧跟在数字后面的位置）。

环视不会“占用”字符

在理解顺序环视和其他环视功能时需要特别注意一点，即在检查子表达式能否匹配的过程中，它们本身不会“占用”任何文本。这可能有点难懂，所以我准备了下面的例子。正则表达式「Jeffrey」匹配：

…by Jeffrey Friedl.

但同样的正则表达式，如果使用顺序环视功能，即「(?=Jeffrey)」，则匹配标记的位置：

…by ▲Jeffrey Friedl.

顺序环视会检查子表达式能否匹配，但它只寻找能够匹配的位置，而不会真正“占用”这些字符。不过，把顺序环视和真正匹配字符的部分——例如「Jeff」——结合起来，我们能得到比单纯的「Jeff」更精确的结果。结合之后的正则表达式是「**(?=Jeffrey)**Jeff」，下一页的图说明，它只能匹配“Jeffrey”这个单词中的“Jeff”。它能够匹配：

…by Jeffrey Friedl.

在此处它的匹配和单纯的「Jeff」一样，但是下面的情况不会匹配：

…by Thomas Jefferson

「Jeff」自己能够匹配这一行，但是因为不存在「(?=Jeffrey)」能够匹配的位置，整个表达式就无法匹配。现在环视的好处还看得不是很明显，但是请不用担心，现在我们只需要关心顺序环视的原理——我们很快会遇到能够充分展现其价值的例子,。

受此启发，你或许会发现「**(?=Jeffrey)**Jeff」和「Jeff**(?=rey)**」是等价的（能够发现这一点的读者很了不起）。它们都能匹配“Jeffrey”这个单词中的“Jeff”。

我们还需要认识到，它们结合的顺序非常重要。「Jeff**(?=Jeffrey)**」不会匹配上面的任何一个例子，而只会匹配后面紧跟有“Jeffrey”的“Jeff”。

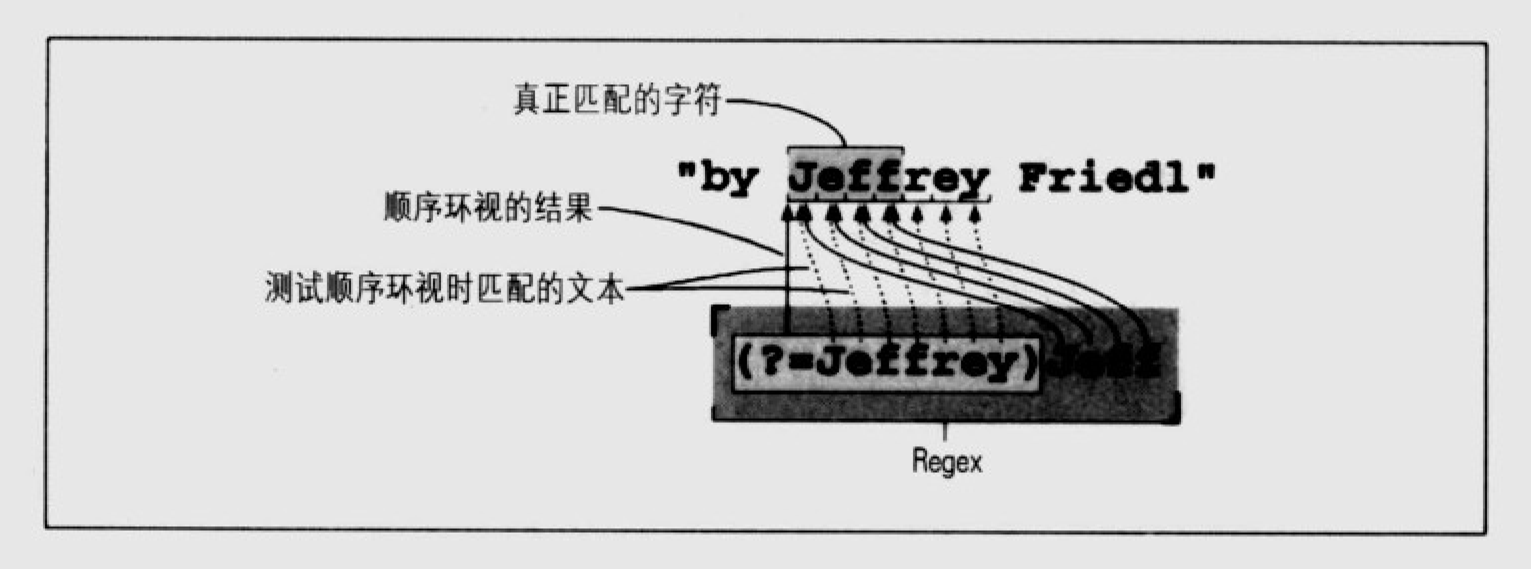


图 2-5：「(?=Jeffrey)Jeff」的匹配

还有一点很重要，即环视结构使用特殊的表示法。就像 45 页介绍的非捕获型括号“(?:…)”一样，它们使用特殊的字符序列作为自己的“开括号”。这样的“开括号”序列有许多种，但它们都以两个字符“(?”开头。问号之后的字符用来标志特殊的功能。我们曾经看到过“分组但不捕获”的“(?:…)”、顺序环视的“(?=…)”，以及逆序环视的“(?<=…)”结构，下面还会看到更多。

再来几个顺序环视的例子

我们马上就要在数字间插入逗号了，不过现在先多看几个环视的例子。首先我们要把所有格“Jeffs”替换为“Jeff's”。不使用环视也能很容易做到这一点，即 **s/**Jeffs**/**Jeff's**/g**（记住，/g 表示“全局替换”，☞51）。更好的办法是添加单词分界符锚点：**s/**\bJeffs\b**/**Jeff's**/g**。

我们也可以使用更复杂的表达式，例如 **s/**\b(Jeff)(s)\b**/**$1'$2**/g**，但是这样简单的任务似乎不值得这么麻烦，所以我们暂时仍然使用 **s/**\bJeffs\b**/**Jeff's**/g**。现在来看另一个正则表达式

```
s/\bJeff(?=s\b)/Jeff'/g
```

两者唯一的区别在于，最后的「s\b」现在位于顺序环视结构。下一页的图 2-6 说明了这个正则表达式的匹配情况。正则表达式变化之后，replacement 字符串中的‘s’也相应地被删去了。

「Jeff」匹配之后，接下来尝试的就是顺序环视。只有当「s\b」在此位置能够匹配时（也就是‘Jeff’之后紧跟一个‘s’和一个单词分界符）整个表达式才能匹配成功。但是，因为「s\b」只是顺序环视子表达式的一部分，所以它匹配的‘s’不属于最终的匹配文本。记住，「Jeff」确定匹配文本，而顺序环视只是“选择”一个位置。在此处使用顺序环视的唯一好处在于，

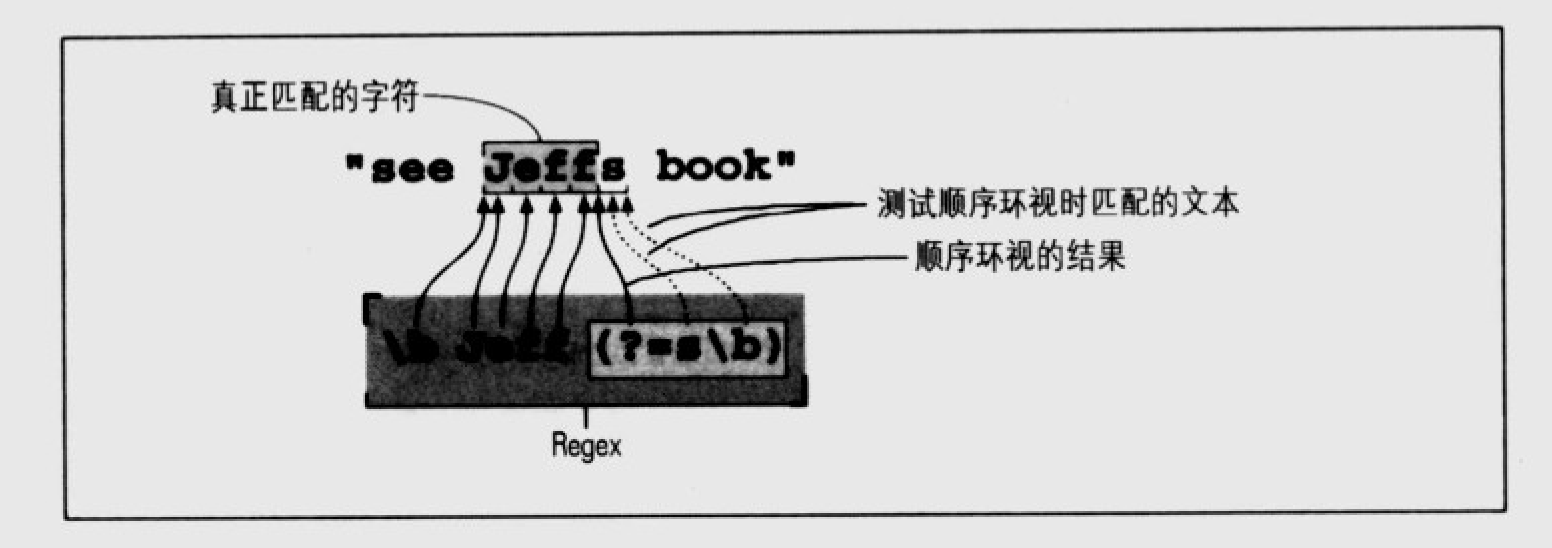


图 2-6：「\bJeff(?=s\b)」的匹配

它保证表达式不会匹配任意的情况。或者从另一个角度来说就是，它容许我们在只匹配「Jeff」之前检查整个「Jeffs」。

为什么不在最终匹配的结果中包含顺序环视匹配过的文本呢？通常，这是因为我们希望在表达式的后面部分，或者在稍后应用正则表达式时，再次检测这段文本。过几页，当我们真正开始解决在数值中加入逗号的问题时，就会明白它的作用。但是在上面的例子中，使用顺序环视的原因在于：我们希望检查整个「Jeffs」，因为这是我们希望加入撇号的地方，但是如果匹配的只是‘Jeff’，就能减小 replacement 字符串的长度。因为‘s’不再是最终匹配结果的一部分，也就不再是 replacement 的一部分，所以我们可以从 replacement 字符串中去掉它。

所以，尽管这两种办法所用的正则表达式和 replacement 字符串各不相同，它们的结果却是一样的。现在看起来，这些应用正则表达式的技巧都有些花架子的味道，但是我这么做是有目的的，请继续往下看。

比较上面的两个例子，最后的「s」从“主（main）”表达式中移到了顺序环视部分中。如果我们把开头的「Jeff」照样搬到逆序环视中呢？结果是「(?<=\bJeff)(?=s\b)」，它的意思是，找到这样一个位置，它紧接在‘Jeff’之后，在‘s’之前。这正好就是我们希望插入撇号的地方。所以，我们这样替换：

```
s/(?<=\bJeff)(?=s\b)/'/g
```

这个表达式很有意思，它实际上并没有匹配任何字符，只是匹配了我们希望插入撇号的位置。在这种情况下，我们并没有“替换”任何字符，而只是插入了一个撇号。图 2-7 作了说明。在几页以前，我们看到过这样的替换，使用 s/^/|>·/在行首加入‘|>·’。

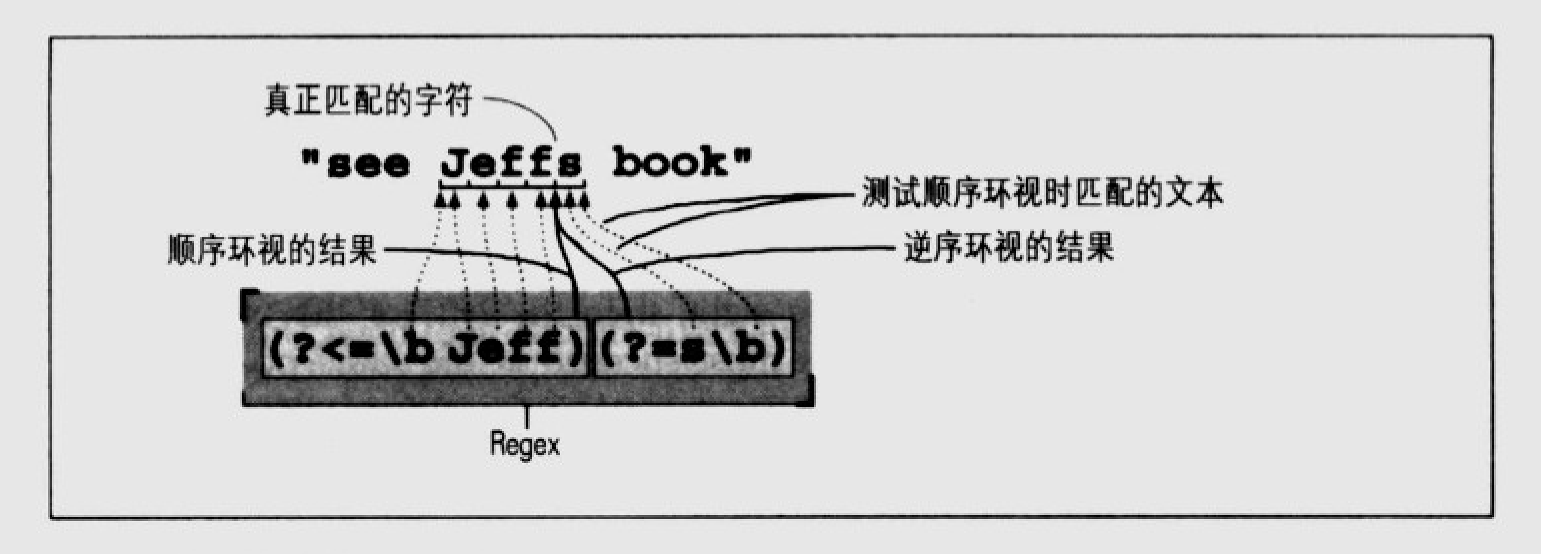


图 2-7：「(?<=\bJeff)(?=s\b)」的匹配

如果我们把两个环视结构调换位置，这个正则表达式的功能会改变吗？也就是说，`s/(?=s\b)(?<=\bJeff)/'/g`的结果如何？❖请翻到下一页查看答案。

"*Jeffs*" 匹配总结 表 2-1 总结了我们见过的把`Jeffs`替换为`Jeff's`的几种办法。

表 2-1：解决"Jeffs"问题的几种办法

解决方案	评　价
`s/\bJeffs\b/Jeff's/g`	最简单，最直接，效率高；解决此类问题最容易想到的办法，未使用环视，正则表达式"占用"整个'Jeffs'。
`s/\b(Jeff)(s)\b/$1'$2/g`	只增加了复杂程度，没有好处，正则表达式占用整个'Jeffs'。
`s/\bJeff(?=s\b)/Jeff'/g`	并没有占用's'，除了展示顺序环视之外，没什么实用价值。
`s/(?<=\bJeff)(?=s\b)/'/g`	并没有"占用"任何文本，同时使用顺序环视和逆序环视匹配需要的**位置**，即撇号插入的位置。非常适于讲解环视。
`s/(?=s\b)(?<=\bJeff)/'/g`	与上一个表达式完全相同，只是颠倒了两个环视结构。因为它并没有占用任何字符，所以变换顺序并没有影响。

回到"在数值中加入逗号"之前，我先提一个关于这些表达式的问题。如果我希望找到不区分大小写的"Jeffs"，在替换之后仍然保持原来的大小写，使用`/i`能实现这个目标吗？

测验答案

◆ 63 页小测验的答案

s/(?=s\b)(?<=\bJeff)/'/g 的结果是什么？

在本例中，「(?=s\b)」和「(?<=\bJeff)」的先后顺序是无关紧要的。无论是“先检查左边，再检查右边”还是相反，关键是，在同一个位置两边的检测必须都能成功，整个匹配才算成功。例如，在字符串‘Thoma‸s•Jeff‸erson’中，「(?=s\b)」和「(?<=\bJeff)」都能匹配（在标记的两个位置），但是这两个位置并不重合，所以这两个环视的结合体不能成功匹配。

“两者的结合”（combanation of the two）虽然有些模糊，但用来描述这个问题并没有问题，因为在此处它的意义很明显。不过，有时候，正则引擎应用表达式的方式可能并不这么明显。因为引擎的工作原理对正则表达式的实际意义有直接的影响，详细讲解见第4章。

提示：至少有两个表达式无法做到这一点。◆请思考这个问题，答案见下页。

回到逗号的例子…

你可能已经意识到了“*Jeffs*”的例子和插入逗号的例子之间存在某种联系，因为它们都需要通过正则表达式寻找到**某个位置**，然后插入文本。

我们已经知道我们希望插入逗号的位置必须满足“左边有数字，右边数字的个数正好是 3 的倍数”。第二个要求用逆序环视很容易解决，左边只要有一位数字就能够满足“左边有数字”的要求，这就是「(?<=\d)」。

现在来看“右边数字的个数正好是 3 的倍数”。3 位数字当然可以表示为「\d\d\d」，我们可以用「(…)+」来表示（3 的）“若干倍”，再添加一个「$」来确保这些数字后面不存在其他字符（保证“正好”）。孤立的「(\d\d\d)+$」匹配从字符串末尾向前数的 3x 位数字，但是加入「(?=…)」的环视结构之后，它就能匹配“右边数字的个数正好是 3 的倍数的**位置**”，例如‘‸123‸456‸789’中的标记位置。实际上并不是所有这些位置都符合要求——我们不希望在第一个数字之前加入逗号——所以我们添加「(?<=\d)」来限定匹配的位置。

代码段如下：

```
$pop =~ s/(?<=\d)(?=(\d\d\d)+$)/,/g;
print "The US population is $pop\n";
```

确实输出了我们期望的“The US population is 298,444,215”。不过，有点奇怪的是，「\d\d\d」两边的括号是捕获型括号。但是在这里，我们只用它来分组，把加号作用于 3 位数字，所以不需要把它们捕获的文本保存到$1 中。

我可以使用第 45 页补充内容介绍的非捕获型括号：「(?:…)」，得到「(?<=\d)(?=(?:\d\d\d)+$)」。这样做的好处在于，见到这个正则表达式的人不会担心与捕获型括号关联的$1 是否会被用到；而且它的效率更高，因为引擎不需要记忆捕获的文本。另一方面，即使是「(…)」也有点难以看懂，更不用说「(?…)」了，所以我在这里选择更清晰的表达方式。构建正则表达式时，经常需要权衡这两个因素。从我个人来说，我愿意在适用的所有地方使用「(?:…)」，但是在讲解其他知识时选择更清晰的表达方式（也是本书中的常见情况）。

单词分界符和否定环视

现在假设，我们希望把这个插入逗号的正则表达式应用到很长的字符串中，例如：

```
$text = "The population of 298444215 is growing";

……

$text =~ s/(?<=\d)(?=(\d\d\d)+$)/,/g;
print "$text\n";
```

很显然程序没有结果，因为「$」要求字符串以 3 的倍数位数字结尾。我们不能只去掉这里的「$」，因为这样会从左边第一位数字之后，右边第三位数字之前的每一个位置插入逗号——结果是“… of 2,9,8,4,4,4,215 …”!

可能初看起来这问题有些棘手，但我们可以用单词分界符「\b」来替换「$」。尽管我们处理的只是数字，Perl 的“单词”概念也能够解决这个问题。就像「\w」(☞49) 一样，Perl 和其他语言都把数字、字母和下画线当作单词的一部分。结果，单词分界符的意思就是，在此位置的一侧是单词（例如数字），另一侧不是（例如行的末尾，或者数字后面的空格）。

这个“一侧如此这般，另一侧如此那般”听起来很耳熟，对吗？因为这正是我们在“Jeffs”例子中所做的。区别之一在于，有一侧必须使用否定的匹配。这样看来，迄今为止我们用到的顺序环视和逆序环视应该被称作**肯定顺序环视**（positive lookahead）和**肯定逆序环视**

测验答案

◆ 第64页的测验答案

在使用/i时，哪些办法会保留原来文本的大小写？

为了保留大小写，要么仅仅替换被占用的字符（而不是任何情况下都插入‘Jeff's’），要么不占用任何字符。表2-1中的第二个表达式采取了第一种思路，匹配占用的字符，用$1和$2把它们替换回来。后两种解法选择了“不占用任何字符”的思路。因为它们不占用任何字符，所以不需要保留任何文本。

第一和第三种解法使用硬编码的replacement字符串。如果使用/i，他们不会保留原来的大小写信息。它们分别把JEFFS错误地替换为Jeff's和Jeff'S。

（positive lookbehind）。因为它们成功的条件是子表达式在这些位置能够匹配。表2-2告诉我们，正则表达式还提供了相对应的**否定顺序环视**和**否定逆序环视**。从名字就能看出，它们成功的条件是子表达式**无法**匹配。

表2-2：四种类型的环视

类型	正则表达式	匹配成功的条件…
肯定逆序环视	(?<=......)	子表达式能够匹配**左侧**文本
否定逆序环视	(?<!......)	子表达式不能匹配**左侧**文本
肯定顺序环视	(?=......)	子表达式能够匹配**右侧**文本
否定顺序环视	(?!......)	子表达式不能匹配**右侧**文本

所以，如果单词分界符的意思是：一侧是「\w」而另一侧不是「\w」，我们就能用「(?<!\w)(?=\w)」来表示单词起始分界符，用「(?<=\w)(?!\w)」表示单词结束分界符。把两者结合起来，「(?<!\w)(?=\w)|(?<=\w)(?!\w))」就等价于「\b」。在实践中，如果语言本身支持\b（\b更直接，效率也更高），这样做有点多此一举，但是可能的确有地方需要用到这两个单独的多选分支（☞134）。

对我们的逗号插入问题来说，我们真正需要的是「(?!\d)」来标记3位数字的起始计数位置。我们用它来取代「\b」或者「$」，得到：

```
$text =~ s/(?<=\d)(?=(\d\d\d)+(?!\d))/,/g;
```

这个表达式在处理类似“…tone of 12345Hz”的文本时效果很好；不幸的是，它同样会匹配“…the 1970s …”中的年份。实际上，我们根本不希望这里的正则表达式能够匹配“…in

1970…"。所以，我们必须知道期望用正则表达式处理的文本，以及开发的程序适合解决什么样的问题（如果数据包含年份信息，这个正则表达式可能就不适合）。

在关于单词分界符和不希望匹配的字符的讨论中，我们使用了否定顺序环视，「(?!\w)」或「(?!\d)」。你可能还记得第 49 页出现的表示"非数字"的字符「\D」，认为它可以取代「(?!\d)」。这并不正确。记住，「\D」的意思是，"某个不是数字的字符"，"某个字符"是必须的，只是它不能为数字。如果在搜索的文本中，数字之后没有字符，「\D」是无法匹配的（在第 12 页的补充内容中我们见到过类似的情况）。

不通过逆序环视添加逗号

逆序环视和顺序环视一样，所获的支持十分有限（使用也不广泛）。顺序环视比逆序环视早出现几年，尽管 Perl 现在两者都支持，许多其他语言却不是这样。所以，想一想不用逆序环视来解决添加逗号的问题可能更有意义。来看下面的表达式：

```
$text =~ s/(\d)(?=(\d\d\d)+(?!\d))/$1,/g;
```

它与之前的表达式的差别在于，开头的「\d」所处的肯定逆序环视变成了捕获型括号，replacement 字符串则在逗号之前加入了相应的$1。

如果我们连顺序环视也不用呢？我们可以用「\b」取代「(?!\d)」，但这个消除逆序环视的技巧是否对剩下的顺序环视有效呢？也就是说，下面的办法可行吗？

```
$text =~ s/(\d)((\d\d\d)+\b)/$1,$2/g;
```

◆ 请翻到下页查看答案。

Text-to-HTML 转换

Text-to-HTML Conversion

现在我们写一个把 Text（纯文本）转换为 HTML（超文本）的小工具，如果要处理所有的情况，程序将非常难写，所以现在我们只写一个用于教学的小工具。

在目前我们看过的所有例子中，作为正则表达式应用对象的变量都只包含一行文本。对这个例子来说，把我们需要转换的所有文本放在同一个字符串中比较方便。在 Perl 中，我们可以很容易地这样做：

```
undef $/;        # 进入"file-slurp"（文件读取）模式
$text = <>;      # 读入命令行中指定的第一个文件
```

测验答案

◆ 67 页问题的答案

`$text =~ s/(\d)((\d\d\d)+\b)/$1,$2/g;` 能够在数字中添加逗号吗？

结果并非我们的期望。得到的是类似“281,421906”的字符串。这是因为「`(\d\d\d)+`」匹配的数字属于最终匹配文本，所以不能作为“未匹配的”部分，供`/g`的下一次匹配迭代使用。

一次迭代完成时，下一次的迭代会从上一次匹配的终点开始尝试。我们希望的是，在插入逗号以后，还能够继续检查这个数值，以决定是否需要再插入逗号。但是，在这个例子中，重新开始的起点是整个数值的末尾。使用顺序环视的意义在于，检查某个位置，但检查时匹配的字符并不算在（最终）“匹配的字符串”内。

实际上，这个表达式仍然可以用来解决这个问题，但正则表达式必须由宿主语言反复调用，例如通过一个 while 循环，每次检查的都是上次修改后的字符串。每次替换操作都会添加一个逗号（对目标字符串中的每个数值都是如此，因为`/g` 的存在），下面是一个例子：

```
while ( $text =~ s/(\d)((\d\d\d)+\b)/$1,$2/g ) {
        # 循环内不用进行任何操作——我们希望的是重复这个循环，直到匹配失败
}
```

如果我们的样本文件包含 3 个短行：

```
This is a sample file.
It has three lines.
That's all
```

变量`$text` 的内容就是：

```
This is a sample file.␤It has three lines.␤That's all␤
```

在某些平台上，也可能是：

```
This is a sample file.␍␤It has three lines.␍␤That's all␍␤
```

这是因为大多数系统采用换行符作为一行的终结符，而某些系统（主要是 Windows）使用回车/换行的结合体。我们会确保这个简单的工具能应付这两种情况。

处理特殊字符

首先我们需要确保原始文本中的‘`&`’、‘`<`’和‘`>`’字符“不会出错”，把它们转换为对应的 HTML 编码，分别是‘`&amp`’、‘`&lt`’和‘`&gt`’。在 HTML 中这些字符有特殊的含义，

编码不正确可能会导致显示错误。我称这种简单的转换为"为 HTML 而加工(cooking the text for HTML)"，它的确非常简单：

```
$text =~ s/&/&amp;/g;  # 保证基本的 HTML...
$text =~ s/</&lt;/g;   # ... 字符 &、<、and > ...
$text =~ s/>/&gt;/g;   # ... 转换后不出错
```

请注意，我们使用了/g 来对所有的目标字符进行替换（如果不用/g，就只会替换第一次出现的特殊字符）。首先转换&是很重要的，因为这三者的 replacement 中都有'&'字符。

分隔段落

接下来我们用 HTML tag 中表示分段的<p>来标记段落。识别段落的简单办法就是把空行作为段落之间的分隔。搜索空行的办法有很多，最容易想到的是：

```
$text =~ s/^$/<p>/g;
```

它可以匹配"行末尾紧随行开头的位置"。确实，我们已经在第 10 页看到，在 *egrep* 之类的工具中这样行得通，因为其中被检索的文本通常只包含逻辑上的一行文本。在 Perl 中也同样有效，对于之前看到过的 E-mail 的例子，我们知道每一个字符串只包含一个逻辑行。

但是，我已经在第 55 页的脚注中提到过，「^」和「$」通常匹配的不是逻辑行的开头和结尾，而是整个的**字符串**的开头和结束位置（注 5）。所以，既然目标字符串中有多个逻辑行，就需要采取不同的办法。

幸好，大多数支持正则表达式的语言提供了一个简单的办法，即**"增强的行锚点"**(enhanced line anchor)匹配模式，在这种模式下，「^」和「$」会从字符串模式切换到本例中需要的逻辑行模式。在 Perl 中，使用/m 修饰符来选择此模式：

```
$text =~ s/^$/<p>/mg;
```

请注意这里同时使用了/m 和/g(你可以以任何顺序排列需要使用的多个修饰符)。在下一章，我们会看到其他语言是如何处理修饰符的。

所以，如果我们从$text 的'…chapter.␊␊Thus…'开始，会得到期望的'…chapter.␊**<p>**␊Thus…'。

不过，如果在"空行"中包含空格符或者其他空白字符，这么做就行不通。为了处理空白字符，我们使用「^·*$」，或者是「^[·\t\r]*$」来匹配某些系统在换行符之前的空格符、制表

注 5：实际上，「$」通常比简单的"字符串结尾"要更复杂些，尽管对本例中这并不重要。详细信息请阅读第 129 页关于行结束锚点的讨论。

符或者回车符。这两个表达式与「^$」是完全不同的，因为它们确实匹配了一些**字符**，而「^$」只匹配**位置**。不过，因为在本例中我们不需要这些空格符、制表符和回车符，匹配（然后用分段 tag 来替换）这些字符不会带来任何问题。

如果你还记得第 47 页的「\s」，你可能会想到「^\s*$」，就像我们在第 55 页 E-mail 的例子中所用的那样。如果用「\s」取代「[·\t\r]」，因为「\s」能够匹配换行符，所以整个表达式的意义就不再是"寻找空行及只包括空白字符的行"，而是"寻找连续、空行和只包括空白字符的行的结合"。也就是说，如果我们找到多个连续的这样的文本行，一个「^\s*$」就能够匹配它们。这样的好处在于，只会留下一个<p>，而不是像以前那样有多少空行就留下多少<p>。

所以，如果$text 有这样的字符串：

```
…with.␤␤␤␉·␤Therefore…
```

我们用：

```
$text =~ s/^[ \t\r]*$/<p>/mg;
```

结果就是

```
…with.␤<p>␤<p>␤<p>␤Therefore…
```

不过，如果我们用：

```
$text =~ s/^\s*$/<p>/mg;
```

结果要更好看一些：

```
…with.␤<p>␤Therefore…
```

所以，在最终的程序中，我们会使用「^\s*$」。

将 E-mail 地址转换为超链接形式

Text-to-HTML 转换的下一步是识别出 E-mail 地址，然后把它们转换为"mailto"链接。例如，*jfriedl@oreilly.com* 会被转换为<a·href="mailto:jfriedl@oreilly.com">jfriedl@oreilly.com。

用正则表达式来匹配或者验证 E-mail 地址是常见的情况。E-mail 地址的标准规范异常繁杂，所以很难做到百分之百的准确，但是一些简单的正则表达式就可以应付遇到的大多数 E-mail 地址。E-mail 地址的基本形式是 *username@hostname*。在思考该用怎样的表达式来匹配各个部分之前，我们先看看这个正则表达式的具体应用环境：

```
$text =~ s/\b(username regex\@hostname regex)\b/<a href="mailto:$1">$1<\/a>/g;
```

需要注意的一点是其中两个用下画线标注的反斜线，第一个在正则表达式（'\@'）中，另

一个在 replacement 字符串的末尾。使用这两个反斜线的理由各不相同。我会在稍后讨论\@（☞77），现在我们只需要知道，Perl 规定作为文本字符的@符号必须转义。

先介绍 replacement 字符串中在‘/’之前的反斜线比较好。我们已经看到，Perl 中查找替换的基本形式是 s/*regex*/*replacement*/*modifier*，用斜线来分隔。所以，如果我们需要在某个部分中使用斜线，就必须使用转义，否则反斜线会被识别为分隔符，作为字符串的一部分。也就是说，如果我们希望在 replacement 字符串中使用，就必须写作<\/a>。

这么做当然可以，但不太好看，所以 Perl 容许用户自定义分隔符。例如 s!*regex*!*string*!*modifier*，或者 s{*regex*}{*string*}*modifier*。无论采用哪种形式，因为 replacement 字符串中的斜线不再与分隔符有冲突，也就不需要转义。第二种形式的分隔符非常明显，所以从现在开始我们采用这种形式。

回到程序中来，请注意整个地址是处于「\b…\b」之间的。添加这些单词分界符能够避免不完整匹配的情况，例如‘jfriedl@oreilly.compiler’。尽管遭遇这种无意义的字符串的几率很小，但使用单词分界符来避免此类匹配一点也不麻烦，所以我会这么做。请注意我是如何用括号包围整个 E-mail 地址的，这样我们就能使用 replacement 字符串‘<a·href="mailto:**$1**">**$1**’。

匹配用户名和主机名

现在我们来看匹配邮件地址所需要的**用户名**和**主机名**的正则表达式。主机名，例如 regex.info 或者 www.oreilly.com，它们由点号分隔，以‘com’、‘edu’、‘info’、‘uk’或者其他事先规定的字符序列结尾。匹配 E-mail 地址的最简单的办法是「\w+\@\w+(\.\w+)+」，用「\w+」来匹配用户名，以及主机名的各个部分。不过，实际应用起来，我们需要考虑得更周到一些。用户名可以包含点号和连字符（虽然用户名不会以这两种字符开头）。所以，我们不应该使用「\w+」，而应该用「\w[-.\w]*」。这就保证用户名以「\w」开头，后面的部分可以包括点号和连字符。（请注意，我们在字符组中把连字符排在第一位，这样就确保它们被作为连字符，而不是用来表示范围。对许多流派来说，.-\w 表示的范围肯定是错误的，它会产生一个随机的字母、数字和标点符号的集合，具体取决于程序和计算机所用的字符编码。Perl 能够正确处理.-\w，但是使用连字符时多加小心是个好习惯。）

主机名的匹配要复杂一些，因为点号只能作分隔符，也就是说两个点号之间必须有其他字符。所以在前面那个简单的正则表达式中，主机名部分用「\w+(\.\w+)+」而不是「[\w.]+」。后者会匹配‘..x..’。但是，即使是前者，也能够匹配‘Artichokes 4@1.00’，所以我们需要更细心一些。

一个办法是给出末尾部分的可能序列，跟在「**\w+(\.\w+)*\.(com|edu|info)**」之后（实际上，多选分支应该是 **com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z]**，不过为了简洁起见，我在这里只列出几项）。这样就能容许开头的「\w+」部分，然后是可能出现的「\.\w+」部分，最后才是我们指定的可能结尾。

实际上「\w」也不是很合适。「\w」能够匹配 ASCII 字母和数字，这没有问题，但有些系统中「\w」能够匹配非 ASCII 字母，例如 à、ç、Ξ、Æ。在大多数流派中，下画线也是可以的。但这些字符都不应该出现在主机名中。所以，我们或许应该用「[a-zA-Z0-9]」，或者是「[a-z0-9]」加上/i 修饰符（进行不区分大小写的匹配）。主机名可能包括连字符，所以我们用「[-a-z0-9]」（再次注意，连字符应该放在第一位）。于是我们得到用来匹配主机名的「[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)」。

无论使用什么正则表达式，记住它们应用的情境都是很重要的。「[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)」这个正则表达式本身，可以匹配‘run C:\\startup.command at startup’，但是把它置入程序运行的环境中，我们就能确认，它会匹配我们期望的文本，而忽略不期望的内容。实际上，我会把它放入之前提到的。

```
$text =~ s{\b(username regex\@hostname regex)\b}{<a href="mailto:$1">$1</a>}gi;
```

（这里用了 **s{…}{…}**分隔符，以及/i），但这样就必须折行。当然，Perl 不关心这个问题，也不关心表达式是否美观，但我关心。所以我要介绍/x 修饰符，它容许我们重新编排这个表达式：

```
$text =~ s{
        \b
        # 把邮件地址存入变量$1 …
        (
          username regex
          \@
          hostname regex
        )
        \b
}{<a href="mailto:$1">$1</a>}gix;
```

啊哈，现在看起来大不一样了。语句末尾出现了/x（在/g 和/i 之后），它对这个正则表达

式做了两件简单但有意义的事情。首先，大多数空白字符会被忽略，用户能够以“宽松排列（free-format)”编排这个表达式，增强可读性。其次，它容许出现以#开头标记的注释。

要指出的是，加上/x之后，表达式中的大部分空格符变为“忽略自身”元字符（“ignore me” metacharacter)，而#的意思是“忽略该字符及其之后第一个换行符之前的所有字符”(☞111)。它们不是作为字符组内部的元字符（也就是说，即便使用了/x，这些字符组也不是“随意编排”的）来对待的，而且，同其他元字符一样，如果希望把它们作为普通字符来处理，也可以对它们加以转义。当然，「\s」总是能够匹配空白字符，例如**m/**<a \s+ href=…>**/x**。

请注意，/x只能应用于正则表达式本身，而不是replacement字符串。同样，即使我们现在使用的是s{…}{…}的格式，修饰符接在最后的‘}’之后（例如‘}x’)，但是在文中我们仍然使用“/x”代表“修饰符x”。

综合起来

现在，我们可以把用户名、主机名的部分，以及之前的开发成果结合起来，得到相对完整的程序：

```
undef $/;        # 进入“文件读取”模式
$text =<>;       # 读入命令行中指定的第一个文件名
$text =~ s/&/&amp;/g;          # 把基本的 HTML …
$text =~ s/</&lt;/g;           # … 字符 &、< 和 > …
$text =~ s/>/&gt;/g;           # … 进行 HTML 转义

$text =~ s/^\s*$/<p>/mg;       # 划分段落

# 转换为链接形式 …
$text =~ s{
   \b
   # 把地址保存到$1 …
   (
     \w[-.\w]*                                  # username
     \@
     [-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)  # hostname
   )
   \b
}{<a href="mailto:$1">$1</a>}gix;

print $text; # 最后，显示 HTML 文本
```

所有的正则表达式都应用于同一个包含多行文本的字符串，需要注意的是，只有用于划分段落的正则表达式才使用/m修饰符，因为只有那个正则表达式用到了「^」和「$」。对其他正则表达式使用/m并不会产生影响（只会令看程序的人迷惑)。

把 HTTP URL 转换为链接形式

最后，我们需要识别 HTTP URL，将它变为链接形式。也就是说把“*http://www.yahoo.com*”转变为`<a·href=http://www.yahoo.com/>http://www.yahoo.com/</a>`。

HTTP URL 的基本形式是 *http://hostname/path*，其中的*/path* 部分是可选的。于是我们得到下面的形式：

```
$text =~ s{
      \b
      # 将 URL 保存至$1 ...
      (
        http:// hostname
        (
           / path
        )?
      )
}{<a href="$1">$1</a>}gix;
```

主机部分的匹配可以使用在 E-mail 例子中用过的子表达式。URL 的 path 部分可以包括各种字符，在前一章中我们使用的是「`[-a-z0-9_:@&?=+,.!/~*'%$]*`」（☞25），它包括了除空白字符、控制字符和`<>(){}`之外的大多数 ASCII 字符。

在使用 Perl 解决这个问题之前，我们必须对`@`和`$`进行转义。同样，我会在稍后讲解原因（☞77）。现在，我们来看 hostname 和 path 部分：

```
$text =~ s{
       \b
       # 将 URL 保存至$1 ...
       (
         http:// [-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info) \b # hostname
         (
            / [-a-z0-9_:\@&?=+,.!/~*'%\$]* # path 不一定会出现
         )?
       )
}{<a href="$1">$1</a>}gix;
```

你可能注意了，在 path 之后没有「`\b`」，因为 URL 之后通常都是标点符号，例如本书在 O'Reilly 的 URL 是：

http://www.oreilly.com/catalog/regex3/

如果末尾有「`\b`」，就不能匹配。

也就是说，在实际中，我们需要对表示 URL 结束的字符做一些人为的规定。比如下面的文本：

```
Read "odd" news at http://dailynews.yahoo.com/h/od, and
maybe some tech stuff at http://www.slashdot.com!
```

现在正则表达式能够匹配标注出的文本了，当然末尾的标点显然不应该作为URL的一部分。在匹配英文文本中的URL时，末尾的「[.,?!]」是不应该作为URL的一部分的（这并不是什么规定，而是我的经验，而且大多数时候都有效）。这很简单，只需要在表达式的末尾添加一个表示“除「[.,?!]」之外的任何字符”的否定逆序环视，「(?<![.,?!])」即可。结果就是，在我们匹配到作为URL匹配的文本之后，逆序环视会反过头来看一眼，保证最后的字符符合要求。如果不符合要求，引擎就会重新检查作为URL的字符串，直到最终符合要求为止。也就是强迫去掉最后的标点，让最后的逆序环视匹配成功（在第5章我们会看到另一个解决办法☞206）。

插入之后，我们得到了完整的程序：

```
undef $/;        # 进入“文件读取”模式
$text = <>;      # 读入命令行中指定的第一个文件

$text =~ s/&/&amp;/g;        # 转换基本的HTML ...
$text =~ s/</&lt;/g;         # ... 字符 &、< 和 > ...
$text =~ s/>/&gt;/g;         # ... 进行HTML转义

$text =~ s/^\s*$/<p>/mg;     # 划分段落

# 将E-mail地址转换为链接形式 ...
$text =~ s{
   \b
   # 把地址保存到$1 ...
   (
      \w[-.\w]*                                      # username
      \@
      [-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)      # hostname
   )
   \b
}{<a href="mailto:$1">$1</a>}gix;

# 将HTTP URL转换为链接形式 ...
$text =~ s{
   \b
   # 将URL保存至$1 ...
   (
      http:// [-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info) \b # hostname
         (
            / [-a-z0-9_:\@&?=+,.!/~*'%\$]*    # path 不一定会出现
              (?<![.,?!])                      # 不能以[.,?!]结尾
         )?
   )
}{<a href="$1">$1</a>}gix;
print $text; # 最后，显示结果
```

构建正则表达式库

请注意，在这两个例子中，我们使用同样的正则表达式来匹配主机名，也就是说，如果要修改匹配主机名的表达式，我们希望这种修改会同时对两个例子生效。我们可以在程序中多次使用变量$HostnameRegex，而不是把这个表达式写在各处，杂乱无绪：

```
$HostnameRegex = qr/[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)/i;

# 将E-mail地址转换为链接形式 …
$text =~ s{
   \b
   #将地址保存至 $1 …
   (
     \w[-.\w]*                        # username
     \@
     $HostnameRegex                   # hostname
   )
   \b
}{<a href="mailto:$1">$1</a>}gix;

# 将HTTP URL转换为链接形式 …
$text =~ s{
   \b
   # Capture the URL to $1 …
   (
      http:// $HostnameRegex \b            # hostname
      (
         / [-a-z0-9_:\@&?=+,.!/~*'%\$]*    # path不一定会出现
           (?<![.,?!])                     # 不容许以[.,?!]结尾
      )?
   )
}{<a href="$1">$1</a>}gix;
```

第一行使用了Perl的qr操作符。它与m和s操作符类似，接收一个正则表达式（例如，使用qr/…/，类似使用m/…/和s/…/…/），但并不马上把这个正则表达式应用到某段文本中进行匹配，而是由这个表达式生成为一个“**regex对象**（regex object）”，作为变量保存。之后我们就能使用这个对象。（在本例中，我们用变量$HostnameRegex来保存这个变量，供后面两个正则表达式使用。）这样做非常方便，因为程序看起来非常清楚。此外，我们的匹配主机名的正则表达式只存在一个“主源(main source)”，这样无论在哪里需要匹配主机名，都可以直接使用它。第6章（☞277）还有关于构建这种“正则表达式库”的例子，具体讲解见第7章（☞303）。

其他的语言也提供了创建正则表达式对象的方法，下一章我们会简要介绍若干语言，而Java和.NET则在第8和第9章详细讲解。

为什么有时候$和@需要转义

你可能注意到了，‘$’符号既可以作为表示字符串结束的元字符，又可以用来标记变量。通常，‘$’的意思是很明确的，但如果在字符组内部，情况就有些麻烦，因为此时它不能用来表示字符串的结束位置，只能在转义之后，用来标记变量。在转义之后，‘$’就只是字符组的一部分。而这正是我们所要的，所以我们需要在 URL 匹配的正则表达式中对它进行转义。

@的情况与之类似。Perl 用@表示数组名，而 Perl 中的字符串或正则表达式中也容许出现数组变量。如果我们希望在正则表达式中使用@字符，就需要进行转义，避免把它作为数组名。

一些语言（Java、VB.NET、C、C#、Emacs、awk 等）不支持变量插值（variable interpolation）。有些语言（例如 Perl、PHP、Python、Ruby 和 Tcl）支持变量插值，但是方法各有不同。我们会在下一章详细讲解（☞101）。

回到单词重复问题

That Doubled-Word Thing

我希望第 1 章提到的单词重复问题能够引发读者对于正则表达式的兴趣。在本章的开头我给出了一堆难懂的代码，将其作为解法之一：

```
$/ = ".\n";
while (<>) {
  next if !s/\b([a-z]+)((?:\s|<[^>]+>)+)(\1\b)/\e[7m$1\e[m$2\e[7m$3\e[m/ig;
  s/^(?:[^\e]*\n)+//mg;    # 删除所有未标记的行
  s/^/$ARGV: /mg;          # 在行首增加文件名
  print;
}
```

对 Perl 有了些了解之后，我希望读者至少能够看懂常规的正则表达式应用——其中的<>，三个 s/…/…/，以及 print。不过，其他的部分仍然很难。如果你关于 Perl 的知识全部来自本章（而且关于正则表达式的知识都来自之前的章节），这个例子可能会超出你的理解能力。

不过，如果细致考察起来，我认为这个正则表达式并不复杂。在重读程序之前，我们不妨回过头看看第 1 页的程序规格要求，并尝试运行一次：

```
% perl -w FindDbl ch01.txt
ch01.txt: check for doubled words (such as this this), a common problem with
ch01.txt: * Find doubled words despite capitalization differences, such as with 'The
ch01.txt:the', as well as allow differing amounts of whitespace (space, tabs,
ch01.txt: /\<(1,000,000|million|[illegible] [illegible])/. But alternation can't be
ch01.txt: of this chapter. If you knew [illegible] [illegible] specific doubled word to find (such
```

……

先来看这个Perl的解法，然后我们会看到一个Java的解法，接触另一种使用正则表达式的思路。现在列在下面的程序使用了 **s{*regex*}{*replacement*}*modifier*** 的替换形式，同时使用了 /x 修饰符来提高清晰程度（空间更充裕的时候，我们使用更易懂的 'next unless' 替换 'next if!'）。除去这些，它与本章开头的程序其实就是一模一样的。

示例 2-3：用 Perl 处理重复单词

```
$/  = ".\n"; ❶ # 设定特殊的“块模式”("chunk-mode")；一块文本的终结为点号和换
                 行符的结合体
while (< >)  ❷
{
   next unless s{❸ #  (下面是正则表达式)
         ### 匹配一个单词：
         \b              # 单词的开始位置 ...
         ( [a-z]+ )      # 把读取的单词存储至$1 (和 \1)

         ### 下面是任意多的空白字符和/或 tag
         (                # 把空白保存到 $2
             (?:          #  (使用非捕获型括号)
                \s           # 空白字符  (包括换行符，这样非常方便)
               |             # 或者是
               <[^>]+>       # <TAG>形式的 tag
             )+           # 至少需要出现一次，多次不受限制
         )

         ### 现在再次匹配第一个单词：
         (\1\b)           # \b 保证用来避免嵌套单词的情况，保存到$3
   # (正则表达式结束)
   }
   # 上面是正则表达式．下面是 replacement 字符串，然后是修饰符、/i、/g 和/x
   {\e[7m$1\e[m$2\e[7m$3\e[m]igx; ❹
   s/^(?:[^\e]*\n)+//mg;     ❺  # 去掉所有未标记的行
   s/^/$ARGV: /mg;           ❻  # 在每行开头加上文件名
   print;
}
```

这小段程序中出现了许多我们没见过的东西。下面我会简要地介绍它们以及背后的逻辑，不过我建议读者查看Perl的man page了解细节（如果是正则表达式相关的细节，可以查阅第7章）。在下面的描述中，“神奇”（magic）的意思是“这里用到了读者可能不熟悉的Perl的特性”。

❶ 因为单词重复问题必须应付单词重复位于不同行的情况，我们不能延续在E-mail的例子中使用的普通的按行处理的方式。在程序中使用特殊变量$/（没错，这确实是一个变量）能使用一种神奇的方式，让<>不再返回单行文字，而返回或多或少的一段文字。返回的数据仍然是一个字符串，只是这个字符串可能包含多个逻辑行。

❷ 你是否注意到，<>没有值赋给任何变量？作为 while 中的条件使用时，<>的神奇之处在于，它能够把字符串的内容赋给一个特殊的默认变量（注 6）。该变量保存了 s/…/…/ 和 print 作用的默认字符串。使用这些默认变量能够减少冗余代码，但 Perl 新手不容易看明白，所以我还是推荐，在你习惯之前，把程序写得更清楚一些。

❸ 如果没有进行任何替换，那么替换命令之前的 **next unless** 会导致 Perl 中断处理当前字符串（转而开始下一个字符串）。如果在当前字符串中没有找到单词重复，也就不必进行下一步的工作。

❹ replacement 字符串包含的就是“$1$2$3”，加上插入的 ANSI 转义序列，把两个重叠的词标记为高亮，中间的部分则不标记高亮。转义序列\e[7m 用于标注高亮的开始，\e[m 用于标注高亮的结束（在 Perl 的正则表达式和字符串中，\e 用来表示 ASCII 的转义字符，该字符表示之后的字符为 ANSI 转义序列）。

仔细看看正则表达式中的那些括号，你会发现“$1$2$3”表示的完全就是匹配的文本。所以，除了添加转义序列之外，整个替换命令并没有进行任何实质修改。

我们知道$1 和$3 匹配的是同样的文本（这也是整个程序的意义所在！），所以在 replacement 中只用一个也是可以的。不过，因为这两个单词的大小写可能有区别，我用了两个变量。

❺ 这个字符串可能包括多个逻辑行，不过在替换命令标记了所有的重复单词之后，我们希望只保留那些包含转义字符的逻辑行。去掉不包含转义字符的逻辑行之后，留下的就是字符串中我们需要处理的行。因为我们在替换中使用的是增强的行锚点匹配模式（/m 修饰符），正则表达式「^([^\e]*\n)+」能够找出不包含转义字符的逻辑行。用这个表达式来替换掉所有不需要处理的行。结果留下的只是包含转义字符的逻辑行，也即那些包含单词重复的行（注 7）。

❻ 变量$ARGV 提供了输入文件的名字。结合/m 和/g，这个替换命令会把输入文件名加到留下的每一个逻辑行的开头。多酷！

注 6：默认变量是$_（是的，这也是一个变量）。它可以作为多个函数和操作符的默认操作对象。
注 7：在这里我们假设输入的文本中不包含转义字符。否则程序不能正常工作。

最后，print 会输出字符串中留下的逻辑行以及转义字符。while 循环对输入的所有字符串重复处理（每次处理一段）。

更深入一点：运算符、函数和对象

我之前已经强调过，在本章我以 Perl 作为工具来讲解概念。Perl 的确是一种有用的工具，但我想要强调的是，这个问题利用其他语言的正则表达式解决起来也很容易。

同样，因为 Perl 具有与其他高级语言不同的独特风格，讲解这些概念更加容易。这种独特风格就是，正则表达式是“基础级别（first-class）”的。也就是说，基本的运算符可以直接作用于正则表达式，就好像+和-作用于数字一样。这样减轻了使用正则表达式的“语法包袱”（syntactic baggage）。

其他许多语言并没有这样的特性。因为第 3 章中提到的原因（☞93），许多现代语言坚持提供专用的函数和对象来处理正则表达式。例如，可能有一个函数接收表示正则表达式的字符串，以及用于搜索的文本，然后根据正则表达式能否匹配该文本，返回真值或假值。更常见的情况是，这两个功能（首先对一个作为正则表达式的字符串进行解释（interpretion），然后把它应用到文本当中）被分割为两个或更多分离的函数，就像下一页的 Java 代码一样。这些代码使用 Java1.4 以后作为标准的 java.util.regex 包。

在程序的上部我们看到，在 Perl 中使用的 3 个正则表达式在 Java 中作为字符串传递给 Pattern.compile 程序。通过比较我们发现，Java 版本的正则表达式包含了更多的反斜线，原因是 Java 要求正则表达式必须以字符串方式提供。正则表达式中的反斜线必须转义，以避免 Java 在解析字符串时按照自己的方式处理它们。

我们还应该注意到，正则表达式不是在程序处理文本的主体部分出现，而是在开头的初始化部分出现的。Pattern.compile 函数所作的仅仅是分析这些作为正则表达式的字符串，构建一个“已编译的版本（compiled version）”，将其赋给 Pattern 变量（例如 regex1）。然后，在处理文本的主体部分，已编译的版本通过 regex1.matcher(text) 应用到文本之上，得到的结果用于替换。同样，我们会在下一章探究其中的细节，在这里我们只需要了解，在学习任何一门支持正则表达式的语言时，我们需要注意两点：正则表达式的流派，以及该语言运用正则表达式的方式。

示例 2-4：用 Java 解决重复单词的问题

```
import java.io.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class TwoWord
{
 public static void main(String [] args)
 {
   Pattern regex1 = Pattern.compile(
        "\\b([a-z]+)((?:\\s|\\<[^>]+\\>)+)(\\1\\b)",
        Pattern.CASE_INSENSITIVE);
   String replace1 = "\033[7m$1\033[m$2\033[7m$3\033[m";
   Pattern regex2 = Pattern.compile("^(?:[^\\e]*\\n)+", Pattern.MULTILINE);
   Pattern regex3 = Pattern.compile("^([^\\n]+)", Pattern.MULTILINE);

   // 对于命令行的每个参数进行如下处理……
   for (int i = 0; i < args.length; i++)
   {
     try {
       BufferedReader in = new BufferedReader(new FileReader(args[i]));
       String text;

       // For each paragraph of each file.....
       while ((text = getPara(in)) != null)
       {
           // 应用 3 条替换规则
           text = regex1.matcher(text).replaceAll(replace1);
           text = regex2.matcher(text).replaceAll("");
           text = regex3.matcher(text).replaceAll(args[i] + ": $1");

           // 显示结果
           System.out.print(text);
       }
   }catch (IOException e) {
       System.err.println("can't read ["+args[i]+"]: " + e.getMessage());
      }
   }
}

// 用于读入"一段"文本的子程序
static String getPara(BufferedReader in) throws java.io.IOException
{
  StringBuffer buf = new StringBuffer();
  String line;

  while ((line = in.readLine()) != null &&
         (buf.length() == 0 || line.length() != 0))
  {
     buf.append(line + "\n");
  }
  return buf.length() == 0 ? null : buf.toString();
 }
}
```



第3章 正则表达式的特性和流派概览

Overview of Regular Expression Features and Flavors

现在我们稍微找到点感觉了，也见识了若干使用正则表达式的工具软件，你可能觉得，该坐下来潜心研究研究如何使用它们了。不过，比较比较第1章中不同版本的*egrep*，或是前一章中Perl程序和Java程序的区别就会发现，工具不同，正则表达式的写法和用法都有很大的不同。

在某种特定的宿主语言或工具软件中使用正则表达式时，主要有3个问题值得注意：

- 支持的元字符，以及这些元字符的意义。这通常称为正则表达式的“流派（flavor）”。
- 正则表达式与语言或工具的“交互”（interface）方式。譬如如何进行正则表达式操作，容许进行哪些操作，以及这些操作的目标文本类型。
- 正则表达式引擎如何将表达式应用到文本。语言或工具的设计者实现正则表达式的方法，对正则表达式能够取得的结果有重要的影响。

正则表达式和汽车

购买汽车时，我们需要考虑的问题和上面3点很相似。正则表达式中的元字符是应当首先关注的，它相当于汽车的造型、色彩和内饰，例如CD播放器和真皮座椅。印刷光鲜的宣传册上经常可以见到这些信息，在正则表达式的世界中，与之对应的就是第32页的元字符列表。这些信息很重要，但还不是全部。

正则表达式与宿主语言的交互方式（interface）也很重要。交互方式的一部分内容起到装饰性作用，描述对应的编程语言中正则表达式的应用规则。另一部分内容定义功能，它们决定了语言所能支持的操作，以及操作的难易程度。对应于汽车的例子，它相当于汽车与我们和我们的生活相“结合”的程度。某些问题可能是装饰性的，例如加油口在车的哪一侧，车窗是否能电动升降。其他问题可能重要些，例如是手动变速还是自动变速。还有些关于功能的问题：你的车怎样开进车库？它能装得下一个大号床垫吗？如果是滑雪板呢？或者五个大人？（以及这些人如何进出，在这个问题上四门车显然比两门车有优势）。宣传册会介绍一些此类信息，不过你可能需要阅读封底的小字才能了解所有细节。

最后需要关注的是引擎，以及引擎驱动车轮的原理。汽车的类比在这里不适用，因为大家都理解汽车发动机工作的基本知识：如果是汽油发动机，人们就不会往油箱里加柴油。如果是手动变速，他们不会忘记踩离合器。但是，在正则表达式的世界中，即使是一些最基本的知识：例如正则引擎的匹配原理，以及该原理对表达式的调校和使用的影响，通常都没有文档介绍。但是，这些细节对实际使用正则表达式又极其重要，所以我们会在下一章用整章的篇幅来讲解。

本章的内容

如标题所示，这一章讲解正则表达式的特性和流派。它介绍了经常使用的元字符，以及在具体的工具软件中使用正则表达式的方式。这些内容涵盖了上文提到的前面两点。第三点——正则引擎是如何工作的，这些工作原理有什么实际意义——会在下面的几章中涉及。

关于本章，我要说的一点是，它并不能告诉你某种工具软件中的正则表达式提供了哪些特性，也不会教育你如何使用在提过的各种工具软件和编程语言中运用正则表达式。相反，它的目的是，提供关于正则表达式本身和使用它的工具软件的完整图景。如果我们与世隔绝，只使用一件工具，或许不需要关心其他的工具（或者是该工具的其他版本）有什么差异。但现实情况并非如此，所以了解我们所用工具的技术渊源，或许能够提供有趣而又有价值的启示。



在正则的世界中漫步

A Casual Stroll Across the Regex Landscape

我喜欢在故事的开头讲讲某些正则表达式的流派以及相应程序的演变过程。所以，请准备一杯你最喜欢的热（或凉的）饮料，放轻松，我们一起来看看今天的正则表达式背后古怪的发展史。这样做是为了让你全面了解正则表达式，培养追问“为什么会如此”的习惯。我们为有兴趣的读者准备了一些脚注，不过大部分脚注只能算是博得读者一笑的花絮。

正则表达式的起源

The Origins of Regular Expressions

关于正则表达式，最初的想法来自 20 世纪 40 年代的两位神经学家，Warren McCulloch 和 Walter Pitts，他们研究出一种模型，认为神经系统在神经元层面上就是这样工作的（注 1）。若干年后，数学家 Stephen Kleene 在代数学中正式描述了这种被他称为“**正则集合**”（regular sets）的模型，正则表达式才成为现实。Stephen 发明了一套简洁的表示正则集合的方法，他称之为“正则表达式”（regular expressions）。

20 世纪 50 年代和 60 年代，理论数学界对正则表达式进行了充分的研究。Robert Constable 的文章为那些对数学感兴趣的读者提供了很不错的简介（注 2）。

尽管存在更古老的应用正则表达式的证据，但我能找到的是，关于在计算方面使用正则表达式的资料，最早发表的是 1968 年 Ken Thompson 的文章 *Regular Expression Search Algorithm*（注 3），在文中，他描述了一种正则表达式编译器，该编译器生成了 IBM 7094 的 object 代码。由此也诞生了他的 *qed*，这种编辑器后来成了 Unix 中 *ed* 编辑器的基础。

ed 的正则表达式并不如 *qed* 的先进，但是这是正则表达式第一次在非技术领域大规模使用。*ed* 有条命令，显示正在编辑的文件中能够匹配特定正则表达式的行。该命令“g/*Regular Expression*/p”，读作“Global Regular Expression Print”（应用正则表达式的全局输出）。这个功能非常实用，最终成为独立的工具 *grep*（之后又产生了 *egrep*——扩展的 *grep*）。

注 1：文章的标题是 A logical calculus of the ideas imminent in nervous activity，首次刊载于 *Bulletin of Math. Biophysics 5*(1943)，然后收录于 *Embodiments of Mind*（MIT Press 1965）。这篇文章的开头简要描述了神经细胞的行为方式（你知道神经脉冲的速度在每秒 1 米到 150 米之间吗？），下面就都是各种算式了，反正我是一点也不懂。

注 2：Robert L. Constable，“The Role of Finite Automata in the Development of Modern Computing Theory，”in *The Kleene Symposium*, Eds. Barwise, Keisler, and Kunen (North-Holland Publishing Company, 1980), 61-83。

注 3：*Communications of the ACM*, Vol.11, No. 6, June 1968。

Grep 中的元字符

相比 *egrep*，*grep* 和其他早期工具所支持的元字符相当有限。元字符`*`是受支持的，但是`+`和`?`则不受支持（不支持问号是很严重的缺陷）。*Grep* 中用于捕获元字符的是`\(…\)`，而未转义的括号会当作普通字符（注 4）。*grep* 支持行锚点（line anchors），但方式十分有限。如果`^`出现在正则表达式的开头，它就是匹配行开头的元字符。否则它就不是一个元字符，而只是一个普通的脱字符。同样，`$`只有出现在正则表达式的末尾时才被当作元字符。结果，用户没法使用「`end$|^start`」这样的表达式。不过这不要紧，因为 *grep* 不支持多选结构。

元字符的作用规则也很重要。例如，*grep* 的最大问题或许在于，星号无法用来限定括号内的子表达式，而只能用于限定普通的字符、字符组，或者点号。所以，在 *grep* 中，括号的作用仅限于捕获已匹配的文本，而不能用来进行普通的分组。实际上，某些早期版本的 *grep* 甚至不支持括号嵌套。

Grep 的发展历程

尽管今天的许多系统都有对应的 *grep*，但你会注意到，本书中提到 *grep* 时使用的都是过去时态（译注 1）。过去时对应旧版本所属的流派，它们的历史都超过 30 年了。在这段时间中，技术在不断进步，旧的程序也会加入新的特性，*grep* 也不例外。

在最老版本的 *grep* 之上，AT&T 的贝尔实验室加入了一些新的特性，例如从 *lex* 程序中借鉴来的`\{`*min, max*`\}`。他们还修正了`-y`选项，早期版本的 *grep* 通过`-y`进行不区分大小写的匹配，但此功能并不正常。同时，Berkeley 的人加入了表示单词开头和结束的元字符，把`-y`改为`-i`。不幸的是，星号或其他量词仍然无法作用于括号内的表达式。

Egrep 的发展历程

此时，Alfred Aho（同样是 AT&T 的贝尔实验室）写出了 *egrep*，它提供了第 1 章介绍的各种元字符中的大部分元字符。更重要的是，它以一种全然不同（但总的来说更好）的方式实现了这些功能。不但加上了「`+`」和「`?`」，还容许量词作用于括号内的表达式，这大大增强了 *egrep* 的表达能力。

注 4：历史遗留问题：*ed*（因此也包括 *grep*）使用转义的括号来分组，是因为 Ken Thompson 觉得正则表达式主要用于 C 代码，因此匹配普通括号会比回溯更常用。

译注 1：汉语无法完整地表现时态。

同时，多选结构加入了，行锚点也升级到“基础级别”，可以在正则表达式的任何地方使用。不过，*egrep* 也不够完美——有时候它能匹配，但不会显示结果，而且它缺乏某些当今流行的特性。不过无论如何，它都比 *grep* 有用得多。

其他工具的发展历程

就在 *egrep* 演变的同时，其他程序，例如 *awk*、*lex* 和 *sed*，也在按各自的脚步前进。通常，开发人员会把某个程序中自己喜欢的特性添加到其他程序中。有时候，结果并不尽如人意。例如，如果要在 *grep* 中增加对「+」的支持，就不能直接使用‘+’，因为长期来以来在 *grep* 中‘+’都不是元字符，突然进行这种修改会让大家感到不适应。因为‘\+’可能是 *grep* 的用户在正常情况下不会输入的，把它作为“重现一次或多次”的元字符可能更合适。

有时候，添加新特性也会带来新的 bug。另外一些时候，新添加的特性不久后又被删除了。构成流派的各个细微的方面，几乎都没有什么文档，所以新的工具软件要么形成了自己的流派，要么尝试模仿其他工具，提供“看来相似”的功能。

这一切，加上漫长的发展史，众多的程序员，结果就是巨大的谜局（注 5）。

POSIX——标准化的尝试

诞生于 1986 年的 POSIX 是 Portable Operating System Interface（可移植操作系统接口）的缩写，它是一系列标准，确保操作系统之间的移植性。该标准的某些部分关乎正则表达式和使用他们的传统工具，所以值得我们关注。不过，本书涉及的各种流派无一严格地遵守了所有的相关规定。为了厘清正则表达式的混乱局面，POSIX 把各种常见的流派分为两大类：*Basic Regular Expressions*（BREs）和 *Extended Regular Expressions*（EREs）。POSIX 程序必须支持其中的任意一种。下页的表 3-1 简要介绍了这两种流派的元字符。

POSIX 标准的主要特性之一是 *locale*，它是一组关于语言和文化传统——例如日期和时间的格式、货币币值、字符编码对应的意义等——的设定。locals 的目的在于让程序变得国际化。它们不是正则表达式相关的概念，尽管它们会影响正则表达式的使用。举例来说，工作于

注 5：尤其是你尝试一下子解决所有问题的时候更是如此，我现在对此体会深刻。

表 3-1：POSIX 正则表达式流派概览

正则表达式特性	BREs	EREs
点号、^、$、[…]、[^…]	✓	✓
“任意数目”量词	*	*
+和?量词		+ ?
区间量词	\{*min, max*\}	{*min, max*}
分组	\(…\)	(…)
量词可否作用于括号	✓	✓
反向引用	\1 到\9	
多选结构		✓

Latin-1 编码（也称为“ISO-8859-1”）之中时，à 和 À（分别对应十进制编码 224 和 160）也被认为是“字符”，任何不区分大小写的正则表达式都会认为这两个字符是相等的。

另一个例子是\w，通常用于表示“构成单词的字符”（在很多流派中，它等价于[a-zA-Z0-9_]）。这个特性并不是 POSIX 中必须的，但容许出现。如果支持的话，\w 就能对应 locale 中的所有字母和数字，而不仅仅限于 ASCII 编码的字符和数字。

如果程序支持 Unicode，那么关于 locale 的问题就极大地简化了。Unicode 的详细讨论从 106 页开始。

Henry Spencer 的正则表达式包

同样是在 1986 年，发生了于一件更重要的事情，Henry Spencer 发布了用 C 语言写的正则表达式包，这个包可以毫无困难地置入其他程序中——这在当时具有开创性的意义。每一个使用 Henry 的包的程序——的确存在很多——都属于相同的流派，除非程序的作者费尽周折去修改。

Perl 的发展历程

差不多在同时，Larry Wall 开始开发一种工具，也就是日后的 Perl 语言。他的 *patch* 程序已经大大促进了分布式软件开发（distributed software development），但是 Perl 注定要产生重大的影响。

1987 年 12 月，Larry 发布了 Perl Version 1。Perl 很快引起了关注，因为它糅合了其他语言的众多特性，但指向一个明确的目的：就是我们日常所说的“**实用**（useful）”。

Perl 的特性中值得一提的是，它提供了传统上只有专用工具 *sed* 和 *awk* 才提供的正则表达式操作符——这在通用脚本语言中是个首创。正则引擎的代码来自一个早期的项目——Larry 的新闻阅读器 *rn*（其中的正则表达式代码来自 James Gosling 的 Emacs（注 6））。Perl 的正则流派，用当时的标准衡量是很强大的，但功能不如今天那样齐全。它主要的问题在于，最多只能支持 9 组括号，9 个多选结构，最糟糕的是，括号内不容许出现「|」，也不能进行不区分大小写的匹配，不支持字符组中的\w（完全不支持\d 和\s）。也不支持区间量词{*min*, *max*}。

Perl 2 发布于 1988 年 6 月。Larry 完全放弃了原有的正则表达式代码，而采用了前面提到过的 Henry Spencer 的正则表达式包的增强版。括号的数目仍然只有 9 个，但是括号中可以使用「|」了。\d 和\s 的支持也加了进来，\w 现在可以匹配下画线了，从这时开始，\w 能够匹配 Perl 的变量名中容许出现的字符。此外，字符组之内也可以出现元字符（表示否定的元字符、\D、\W 和\S，也可以支持，但不能使用在字符组内部，而且总在有些情况下无法正常工作）。很重要的一点是，添加了/i 量词，能够进行不区分大小写的匹配。

Perl 3 发布于一年多以后的 1989 年 10 月。它添加了/e 量词，这样极大地增强了替换运算符的能力，同时修正了之前版本中的一些与回溯相关的 bug。也添加了{*min*, *max*}区间量词。虽然很不幸，这些量词不能保证在任何情况下都可以正常工作。还有，这时候 Perl 的正则引擎本不应该停留在只处理 8 位编码数据的水平，但是面对非 ASCII 输入时，会产生无法预料的结果。

Perl 4 的发布是在一年半以后，1991 年 3 月，在接下来的两年间，Perl 4 一直在改进，直到 1993 年 2 月发布最终升级。到此时，之前的 bug 已经修正，原有的限制也被突破（\D 之类可以应用在字符组中，而括号的数目也不再有限制），正则引擎也花了很多功夫来优化，不过真正的突破是在 1994 年。

Perl 5 正式发布于 1994 年 10 月。这一版的 Perl 经历了全面的修整，在各个方面都比原来强上许多。就正则表达式来说，它进行了更多的内部优化，添加了少量元字符（\G 增强了迭

注 6：James Gosling 后来去开发他自己的语言 Java，Java 1.4 提供了一个标准的正则表达式包。第 8 章详细介绍了 Java。

代匹配的能力☞130)、非捕获的括号（☞45)、忽略优先（lazy）的量词（☞141)、顺序环视功能（☞60)，以及/x量词（☞72)（注7)。

这些新增功能的意义并不限于功能本身，更重要的是，这些“新增”的修改使正则表达式本身成为一种强大的编程语言，并为它提供了进一步的发展空间。

新增的非捕获型括号和顺序环视结构都需要新的表达方式。而(…)、[…]、<…>和{…}都已经有了含义，所以Larry采用了我们今天使用的‘(?’表示法。这个表示法并不好看，不过在之前的Perl正则表达式中这是不合规则的组合，所以添加起来完全没有障碍。Larry也预见到，将来可能还需要新增其他的功能，所以他对‘(?’之后的字符做了限制，这样就能保留某些字符，用于将来更多的功能。

之后的各版Perl越来越健壮，错误越来越少，内部优化越来越棒，添加了越来越多的新特性。我相信，本书的第一版也为此做了小小的贡献，因为鄙人研究和测试了正则表达式相关的特性，并将结果告知Larry和Perl Porters group，为改进提供了反馈。

后来添加的新特性包括逆序环视功能（☞60)，“固化”分组（“atomic” grouping ☞139)，和Unicode支持。新添加的条件判断结构更是把正则表达式提升到了一个新的层次(☞140)，它容许用户在正则表达式中进行if-then-else的条件判断和控制。如果这些还不够强大的话，新的结构甚至容许程序员在正则表达式中运行Perl代码，正则表达式和程序代码之间的界限已经不复存在了（☞327)。本书中使用的Perl的版本为5.8.8。

流派的部分整合

具有先见之明的Perl 5完全契合了互联网革命的节拍。Perl的初衷是文本处理，而Web页的生成其实正是文本处理，所以Perl迅速成为了开发Web程序的语言。Perl广受欢迎，其中强大的正则流派也是如此。

其他语言的开发人员当然不会视而不见，最终在某种程度上“兼容Perl”(Perl compatible）的正则表达式包出现了。Tcl、Python、.NET、Ruby、PHP、C/C++都有各自的正则表达式包，Java语言中还有多个正则表达式包。

注7：我写过一篇文章来谈长而复杂的正则表达式，为了保持清晰，我对正则表达式进行了“美观的排版”。Larry见到之后觉得在Perl代码中这样做会很方便，于是就添加了/x。这样说来，其中还有本人的功劳。

另一种形式的整合始于 1997 年（凑巧的是，本书的第一版也在当年面世），当时 Philip Hazel 开发了 PCRE，这是一套兼容 Perl 正则表达式的库，PCRE 的正则引擎质量很高，全面仿制 Perl 的正则表达式的语法和语义。其他的开发人员可以把 PCRE 整合到自己的工具和语言中，为用户提供丰富而且极具表现力（也是众所周知）的各种正则功能。许多流行的软件都使用了 PCRE，例如 PHP、Apache 2、Exim、Postfix 和 Nmap（注 8）。

本书对应的版本

表 3-2 列出了本书中使用的工具和库的版本信息。更老的版本可能功能更少，bug 更多，新的版本则会提供更多的特性，并修正之前的 bug（当然也可能多出新的 bug）。

表 3-2：本书中提到的一些工具的版本

GNU awk 3.1	`java.util.regex`（JDK 1.5，也叫 5.0）	Procmail 3.22
GNU *egrep/grep* 2.5.1	.NET Framework 2.0	Python 2.3.5
GNU Emacs 21.3.1	PCRE 6.6	Ruby 1.8.4
flex 2.5.31	Perl 5.8.8	GNU sed 4.0.7
MySQL 5.1	PHP（preg routine）5.1.4/4.4.3	Tcl 8.4

最初印象

At a Glance

我们用一张表格来比较常见工具软件在几方面的功能，以便理解仍然存在的差异。表 3-3 提供了若干工具软件的正则表达式所属流派在各方面的简要信息。

其他书籍通常在比较各款工具软件时，也会包含表 3-3 之类的表格。但是，这张表只是冰山一角——列出的每一种特性的背后，都有许多重要的知识。

最重要的是程序会不断变化。举例来说，Tcl 以前是不支持反向引用和单词分界符的，但是现在支持。最开始，用来表示单词分界符的是难看的`[:<:]`和`[:>:]`，至今仍是这样，尽管这种表示法已经废弃，取代它的是后来添加的`\m`、`\M`和`\y`（单词起始、单词结束，或者两者皆是）。

同样，*grep* 和 *egrep* 并没有单一的作者，只要愿意，任何人都可以开发，也能修改到符合到作者期望的任何流派。人人都希望按照自己的意愿来，人性就是如此（例如，许多常用工

注 8：PCRE 在下面的地址有免费提供 *ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/*。

表 3-3：若干常用工具的 Flavor 的（非常）简要考察

<table>
<tr><th>特性</th><th>Modern grep</th><th>Modern egrep</th><th>GNU Emacs</th><th>Tcl</th><th>Perl</th><th>.NET</th><th>Sun's Java package</th></tr>
<tr><td>*、^、$、[…]</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr>
<tr><td>? + |</td><td>\? \+ \|</td><td>? + |</td><td>? + \|</td><td>? + |</td><td>? + |</td><td>? + |</td><td>? + |</td></tr>
<tr><td>分组</td><td>\(…\)</td><td>(…)</td><td>\(…\)</td><td>(…)</td><td>(…)</td><td>(…)</td><td>(…)</td></tr>
<tr><td>(?:…)</td><td></td><td></td><td></td><td></td><td>✓</td><td>✓</td><td>✓</td></tr>
<tr><td>单词分界符</td><td></td><td>\<\></td><td>\<\>\b、\B</td><td>\m、\M、\y</td><td>\b、\B</td><td>\b、\B</td><td>\b、\B</td></tr>
<tr><td>\w、\W</td><td></td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr>
<tr><td>反向引用</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr>
</table>

✓表示支持

具的 GNU 版本，比其他版本更强大，也更健壮）。

或许与列出的特性一样重要的是流派之间的许多细微（有些并非细微）差别。从表格来看，Perl、.NET 和 Java 的正则表达式似乎是一样的，而实际情况却远不是这样。针对表 3-3，读者可能提出的问题包括：

- 星号之类的量词能否作用于括号之内的子表达式？
- 点号能否匹配换行符？排除型字符组能否匹配换行符？以上两者能否匹配 NUL 字符？
- 行锚点(line anchor)是名符其实的吗(例如，他们能否识别目标字符串内部的换行符)？它们算正则表达式中的基础级别（first-class）的元字符吗？还是只能应用在某些结构中？
- 字符组内部能出现转义字符吗？字符组内部还容许或不容许出现哪些字符？
- 括号能够嵌套吗？如果是，嵌套的深度是否有限制呢（还有个问题是，一共容许出现多少括号呢）？
- 如果容许反向引用，在进行不区分大小写的匹配时，反向引用能顺利进行吗？在极端的情况下，反向引用的“行为”有意义吗？
- 是否可以出现八进制的转义字符`\123`？如果是，怎么区分它和反向引用呢？十六进制的转义字符呢？这种支持是正则引擎提供的，还是由其他工具提供的？

- \w 只支持数字和字符，还是包括其他字符？（表 3-3 列出的支持\w 的工具对\w 有不同的解释）。不同的单词分界符元字符对构成“单词分界符”的字符的定义不一样，\w 是否与它们保持一致？它们是按照 locale 的定义呢，还是支持 Unicode?

即使表 3-3 这样的介绍这样简单，我们仍然必须记得这些问题。如果你能意识到，在看起来光鲜的外表下面潜藏着许多问题，就容易保持清醒的头脑来应付它们。

在本章开头我们已经提到，许多问题只是语法的差异，但也有许多并非如此。比方说，了解到 *egrep* 的「(Jul|July)」在 GNU Emacs 中必须写成「\(Jul\|July\)」之后，你或许会认为所有的问题都是这样，但事实并非如此。在匹配尝试过程中的语义差异（或者，至少是看起来是在匹配尝试过程中的）通常被忽视，但极其重要的问题是，它也解释了为什么两个看起来一样的表达式会获得截然不同的结果：一个总是匹配‘Jul’，即使目标文本是‘July’。这些看起来毫无区别的语义也解释了，为什么两个顺序相反的正则表达式：「(July|Jul)」和「\(July\|Jul\)」能够取得同样的匹配结果。其实，整个下一章都在讲解这类问题。

当然，一款工具软件能够利用正则表达式实现的功能，通常比它所属的正则流派更重要。例如，就算 Perl 的正则表达式功能不及 *egrep*，在使用正则表达式时，Perl 所具有的简便性却更有价值。我们会在本章逐个介绍各种特性，并在后面各章深入讲解几种编程语言。

正则表达式的注意事项和处理方式

Care and Handling of Regular Expressions

本章开头列出的第二点需要注意的就是：正则表达式的句法规则（syntactic packaging），它告诉应用程序：“嘿，这儿有一个正则表达式，我需要你做这些”。*egrep* 是一个简单的例子，因为正则表达式是作为命令行参数传过去的。其他的“语法诀窍（syntactic sugar）”，例如我在第 1 章坚持使用的单引号，是因为考虑到 *shell*，而不是 *egrep*。复杂的系统，例如程序设计语言中的正则表达式，需要更多的包装，系统才能知道哪些部分是正则表达式，需要如何处理。

下一步是考察我们能够对匹配结果进行的操作。同样，*egrep* 在这方面很简单，因为它做的都是同样的事情（显示包含匹配文本的行），但是，我们在前一章的开头已经说过，真正有

意义的是更复杂的操作。其中最基本的是**匹配**（检查一个正则表达式是否能匹配一个字符串，或者从字符串中提取信息），以及**查找和替换**，根据匹配的结果修改字符串。这些操作可以以多种形式进行，不同的语言对此也有不同的规定。

一般来说，程序设计语言有 3 种处理正则表达式的方式：集成式（integrated）、程序式（procedural）和面向对象式（object-oriented）。在第一种方式中，正则表达式是直接内建在语言之中的，Perl 就是如此。但是在其他两种方式中，正则表达式不属于语言的低级语法。相反，普通的函数接收普通的字符串，把它们作为正则表达式进行处理。由不同的函数进行不同的、关系到一个或多个正则表达式的操作。大多数语言（不包括 Perl）采用的都是这两种方式之一，包括 Java、.NET、Tcl、Python、PHP、Emacs、lisp 和 Ruby。

集成式处理

Integrated Handling

我们已经看过 Perl 的集成式处理方法，例如第 55 页的例子：

```
if ($line =~ m/^Subject: (.*)/i) {
    $subject = $1;
}
```

为清楚起见，我用斜体标注变量名，正则表达式相关的部分则用粗体标注，正则表达式本身用下画线标注。Perl 会把正则表达式「`^Subject:·(.*)`」应用到`$line` 保存的文本中，如果能够匹配，则执行下面的程序段。其中，变量`$1` 代表括号内的子表达式匹配的文本，将它们赋值给`$subject`。

另一个集成式处理的例子是把正则表达式作为配置文件的一部分，例如 *procmail*（Unix 下的一个邮件处理程序）。在配置文件中，正则表达式用于将邮件信息发布到对应的处理程序中。这个例子比 Perl 更简单，因为不需要指明操作对象（邮件信息）。

这两个例子背后的原理要复杂一些。集成式处理方法减轻了程序员的负担，因为它隐藏了一些工作，例如正则表达式的预处理，准备匹配，应用正则表达式，返回结果。省略这些操作减轻了常见任务的完成难度，不过我们之后将会看到，有些情况下，这样处理反而更慢，更复杂。

不过，在深入细节之前，我们先打量打量其他的处理方式，然后再来揭示这些被隐藏的步骤。

程序式处理和面向对象式处理

Procedural and Object-Oriented Handling

程序式处理和面向对象式处理非常相似。这两种方式下，正则功能不是由内建的操作符来提供，而是由普通函数（函数式）或构造函数及方法（面向对象式）来提供的。这种情况下，并没有专属于正则表达式的操作符，只有平常的字符串，普通的函数、构造函数和方法把这些字符串作为正则表达式来处理。

下面几节给出了几个 Java、VB.NET、PHP 和 Python 的例子。

Java 中的正则处理

现在来看“Subject”例子在 Java 中的实现方式，使用 Sun 提供的 `java.util.regex` 包（第 8 章详细介绍 Java）。

```
   import java.util.regex.*; // 这样使用 regex 包中的类更加容易
     ......
❶  Pattern r = Pattern.compile("^Sujbcet: (.*)", Pattern.CASE_INSENSITIVE);
❷  Matcher m = r.matcher(line);
❸  if (m.find()) {
❹      subject = m.group(1);
   }
```

我仍然用斜体标注变量名，粗体标注正则表达式相关的元素，下画线标注正则表达式本身。准确地说，是用下画线标注表示作为正则表达式处理的普通的字符串。

这个类说明了面向对象式处理方法，它使用 Sun 提供的 `java.util.regex` 包的两个类——`Pattern` 和 `Matcher`。其中执行的操作有：

❶ 检查正则表达式，将它编译为能进行不区分大小匹配的内部形式（internal form），得到一个“`Pattern`”对象。

❷ 将它与欲匹配的文本联系起来，得到一个“`Matcher`”对象。

❸ 应用这个正则表达式，检查之前与之建立联系的文本，是否存在匹配，返回结果。

❹ 如果存在匹配，提取第一个捕获括号内的子表达式匹配的文本。

任何使用正则表达式的语言都需要进行这些操作，或是显式的（explicitly）或是隐式的（implicitly）。Perl 隐藏了大多数细节，Java 的实现方式则暴露这些细节。

函数式处理的例子。不过，Java 也提供了一些函数式处理的“便捷函数（convenience functions）”来节省工作量。用户不再需要首先声称一个正则表达式对象，然后使用该对象的方法来操作。下面的静态函数提供了临时对象，执行完之后，这些对象就会被自动抛弃。

这个例子用来说明 Pattern.matches(…) 函数：

```
if (! Pattern.matches("\\s*", line))
{
    // ... 如果 line 不是空行 ...
}
```

这个函数包装了一个隐式的「^…$」的正则表达式，返回一个 Boolean 值，说明它是否能够匹配输入的字符串。Sun 的 package 同时提供程序式和面向对象式的处理方式是常见的做法。两种接口的差别在于便捷程度（程序式处理方式在完成简单任务时更容易，但处理复杂任务则很麻烦）、功能（程序式处理方式的功能和选项通常比对应的面向对象式的要少）和效率（在任何情况下，两类处理方式的效率都不同——第 6 章详细论述这个问题）。

Sun 有时也会把正则表达式整合到 Java 的其他部分，例如上面的例子可以使用 string 类的 matches 功能来完成：

```
if (! line.matches("\\s*", ))
{
    // ... 如果 line 不是空行 ...
}
```

同样，这种办法不如合理使用面向对象的程序有效率，所以不适宜在对时间要求很高的循环中使用，但是"随手（casual）"用起来非常方便。

VB 和.NET 语言中的正则处理

尽管所有的正则引擎都能执行同样的基本操作，但即使是采用同样方法的各种实现方式（implementation）提供给程序员完成的任务，以及使用服务的方式也各有不同。下面是 VB.NET 中的"Subject"例子（.NET 在第 9 章详细论述）：

```
Imports System.Text.RegularExpressions ' 这样访问正则表达式的类会更方便

  ......

Dim R as Regex = New Regex("^Subject: (.*)", RegexOptions.IgnoreCase)
Dim M as Match = R.Match(line)
If M.Success
    subject = M.Groups(1).Value
End If
```

总的来说，它很类似 Java 的例子，只是.NET 将第❷和第❸步结合为一步，第❹步需要一个确定的值。为什么会有这样的差异？两者并没有本质上的优劣之分——只是开发人员采用了自己当时觉得最好的方式（稍后我们会看到这点）。

.NET 同样提供了若干程序式处理的函数。下面的代码用于判断空行：

```
If Not Regex.IsMatch(Line, "^\s*$") Then
   ' ... 如果 line 不是空行 ...
End If
```

Java 的 Pattern.matches 函数会自动在正则表达式两端添加「^…$」，微软则提供了更为一般的函数。Java 的做法只是对核心对象的简单包装，但程序员需要使用的字符和变量更少，而代价只是一点点性能下降。

PHP 中的正则处理

下面是使用 PHP 的 preg 套件中的正则表达式函数处理「Subject」的例子，这是纯粹的函数式方法（第 10 章详细介绍 PHP）。

```
if (preg_match('/^Subject: (.*)/i', $line, $matches))
     $Subject = $matches[1];
```

Python 中的正则处理

最后我们来看 Python 中「Subject」的例子，Python 采用的也是面向对象式的办法。

```
import re;
   ......
R = re.compile("^Subject: (.*)", re.IGNORECASE);
M = R.search(line)
if M:
    subject = M.group(1)
```

这个例子与我们之前看过的非常类似。

差异从何而来

为什么不同的语言采用不同的办法呢？可能有语言本身的原因，不过最重要的因素还是正则软件包的开发人员的思维和技术水准。举例来说，Java 有许多正则表达式包，因为这些作者都希望提供 Sun 未提供的功能。每个包都有自己的强项和弱项，不过有趣的是，每个软件包的功能设定都不一样，所以 Sun 最终决定自己提供正则表达式包。

另一个关于这种差异的例子是 PHP，PHP 包含了三种完全独立的正则引擎，每一种都对应一套自己的函数。PHP 的开发人员在开发过程中，因为对原有的功能不满意，添加新的软件包和对应的接口函数套件来升级 PHP 核心（一般认为，本书讲解的“preg”套件是最优秀的）。

查找和替换

A Search-and-Replace Example

"Subject"的例子太简单，还不足以说明 3 种方法之间的差异。在本节我们将看到更复杂的例子，它进一步揭示了不同处理方式在设计上的差异。

在前一章，我们看到了在 Perl 中利用查找和替换将 E-mail 地址转换为超链接的例子（☞73）：

```
$text =~ s{
    \b
    # 把捕获的地址保存到$1 ...
    (
      \w[-.\w]*                              # username
      @
      [-\w]+(\.[-\w]+)*\.(com|edu|info) # hostname
    )
    \b
 }{<a href="mailto:$1">$1</a>}gix;
```

Perl 的查找和替换操作符是"原地生效"的，也就是说，替换会在目标变量上进行。其他大多数语言的替换都是在目标文本的**副本**上进行的。如果不需要修改原变量，这样操作就很方便，不过如果需要修改原变量，就得把替换结果回传给原变量。下面给出了一些例子。

Java 中的查找和替换

下面是使用 Sun 提供的 `java.util.regex` 进行查找-替换的例子：

```
import java.util.regex.*; // 一次性导入所有需要用到的类
  ......
Pattern r = Pattern.compile(
   "\\b                                                        \n"+
   "# 把捕获的地址保存到$1 ...                                 \n"+
   "(                                                          \n"+
   "  \\w[-.\\w]*                          # username          \n"+
   "  @                                                        \n"+
   "  [-\\w]+(\\.[-\\w]+)*\\.(com|edu|info)  # hostname         \n"+
   ")                                                          \n"+
   "\\b                                                        \n",
   Pattern.CASE_INSENSITIVE|Pattern.COMMENTS);
Matcher m = r.matcher(text);
text = m.replaceAll("<a href=\"mailto:$1\">$1</a>");
```

请注意，字符串中的每个'\'都必须转义为'\\'，所以，如果我们像本例中一样用文本字符串来生成正则表达式，「\w」就必须写成'\\w'。在调试时，**`System.out.println(r.pattern())`**可以显示正则函数确切接收到的正则表达式。我在这个正则表达式中包括换行符的原因是，这样看起来很清楚。另一个原因是，每个#引入一段注释，直到该行结束，所以，为了约束注释，必须设定某些换行符。

Perl 使用/g、/i、/x 之类的符号来表示特殊的条件（这些修饰符分别代表**全局替换、不区分大小写和宽松排列模式**☞135），java.util.regex 则使用不同的函数（replaceAll 而不是 replace），以及给函数传递不同的标志位（flag）参数（例如 Pattern.CASE_INSENSITIVE 和 Pattern.COMMENTS）来实现。

VB.NET 中的查找和替换

VB.NET 的程序与 Java 的类似：

```
Dim R As Regex = New Regex _
 ("\b                                                        " & _
  " (?# 将捕获的地址保存到 $1 ... )                          " & _
  "(                                                         " & _
  "  \w[-.\w]*                          (?# username)        " & _
  "  @                                                       " & _
  "  [-\w]+(\.[-\w]+)*\.(com|edu|info)  (?# hostname)        " & _
  ")                                                         " & _
  "\b                                                        ", _
  RegexOptions.IgnoreCase  Or RegexOptions.IgnorePatternWhitespace)

text = R.Replace(text, "<a href=""mailto:${1}"">${1}</a>")
```

因为 VB.NET 的字符串文字（literal）不便于操作（它们不能跨越多行，也很难在其中加入换行符），长一点的正则表达式使用起来不如其他语言方便。另一方面，因为‘\’不是 VB.NET 中的字符串的元字符，这个表达式看起来要更清楚些。双引号是 VB.NET 字符串中的元字符，为了表示这个字符，我们必须使用两个紧挨着的双引号。

PHP 中的查找和替换

下面是 PHP 中的查找和替换的例子：

```
$text = preg_replace('{
                     \b
                     #把捕获的地址保存到 $1 ...
                     (
                       \w[-.\w]*                           # username
                       @
                       [-\w]+(\.[-\w]+)*\.(com|edu|info)   # hostname
                     )
                     \b
                   }ix',
                   '<a href="mailto:$1">$1</a>', # replacement 字符串
                   $text);
```

就像 Java 和 VB.NET 一样，查找和替换操作的结果必须回传给$text，除去这一点，这个例子和 Perl 的很相似。

其他语言中的查找和替换

Search and Replace in Other Languages

下面我们简要看看其他传统工具软件和语言中查找和替换的例子。

Awk

Awk 使用的是集成式处理方法，*/regex/*，来匹配当前的输入行，使用“**var ~ …**”来匹配其他数据。你可以在 Perl 中看到这种匹配表示法的影子（不过，Perl 的替换操作符模仿的是 *sed*）。Awk 的早期版本不支持正则表达式替换，不过现在的版本提供了 sub(…) 操作符。

```
sub(/mizpel/, "misspell")
```

它会把正则表达式「mizpel」应用到当前行，将第一个匹配替换为 misspel。请注意，在 Perl（和 sed）中的对应做法是 **s/mizpel/misspell/**。

如果要对该行的所有匹配文本进行替换，Awk 使用的不是/g 修饰符，而是另一个运算符：**gsub(/mizpel/, "misspell")**。

Tcl

Tcl 采用的是程序式处理方法，对不熟悉 Tcl 引用惯例（quoting conventions）的人来说可能很迷惑。如果我们要在 Tcl 中修正错误的拼写，可以这样：

```
regsub mizpel $var misspell newvar
```

它会检查变量 var 中的字符串，把「mizpel」的第一处匹配替换为 misspell，把替换后的字符串存入变量 newvar（这个变量并没有以$开头）。Tcl 接收的第一个参数是正则表达式，第二个参数是目标字符串，第三个是 replacement 字符串，第四个是目标变量的名字。Tcl 的 regsub 同样可以接收可能出现的标志位，例如-all 用来进行全局替换，而不是只替换第一处匹配文本。

```
regsub -all mizpel $var misspell newvar
```

同样，-nocase 选项告诉正则引擎进行不区分大小写的匹配（它等于 *egrep* 的-i 参数，或者 Perl 的/i 修饰符）。

GNU Emacs

GNU Emacs（下文中简称 Emacs）是功能强大的文本编辑器，它可以使用 *elisp*（Emacs lisp）作为内建的编程语言。它提供了正则表达式的程序式处理接口，以及数量众多的函数来提供各种服务。其中主要的一种是“正则表达式搜索-前进（re-search-forward）”，接收参数为普通字符串，将它作为正则表达式来处理。然后从文本的“当前位置”开始搜索，直

到第一处匹配发生，或者如果没有匹配，就一直前进到字符串的末尾（用户调用编辑器的“正则表达式搜索（regexp search）”的功能时，就会执行 `re-search-forward`）。

如表 3-3（☞92）所示，Emacs 所属的正则流派严重依赖反斜线。例如，「`\<\([a-z]+\)\([\n·\t]\|<[^>]+>\)+\1\>`」是查找重复单词的表达式，可以用来解决第 1 章的问题。但我们不能直接使用这个正则表达式，因为 Emacs 的正则引擎不能识别`\t` 和`\n`。不过 Emacs 中的双引号字符串则可以，它会把这些标记转换为我们需要的制表符和换行符，传给正则引擎。在使用普通字符串提交正则表达式时，非常有用。但其缺陷——尤其是 *elisp* 的正则表达式的缺陷——在于，此流派过分依赖反斜线了，最终得到的正则表达式好像插满了牙签。下面是查找下一组重复单词的函数：

```
(defun FindNextDbl ()
  "move to next doubled word, ignoring <···> tags" (interactive)
  (re-search-forward "\\<\\([a-z]+\\)\\([\n \t]\\|<[^>]+>\\)+\\1\\>")
)
```

这段程序加上**`(define-key global-map "\C-x\C-d" 'FindNextDbl)`**，就可以使用“Control-x+ Control-d”来迅速查找重复单词了。

注意事项和处理方式：小结

Care and Handling: Summary

我们已经看到，函数很多，内部的机制也很多。如果你不熟悉这些语言，可能现在还有些困惑。不过请不必担心。学习任何特定的工具软件都比学习原理要容易。

字符串，字符编码和匹配模式

Strings, Character Encodings, and Modes

在深入讲解常见的各类元字符之前，还需要了解一些重要的问题：作为正则表达式的字符串，字符编码和匹配模式。

这些概念并不复杂，在理论和实践中都是如此。不过，对其中的大多数来说，因为各种实现方式之间存在细小差异，我们很难预先知道它们准确的实际使用方式。下一节涵盖了若干你将面对的常见问题，以及一些复杂的问题。

作为正则表达式的字符串

Strings as Regular Expressions

这个概念并不复杂：对除 Perl、*awk*、*sed* 之外的大多数语言来说，正则引擎接收的是以普通字符串形式提供的正则表达式，这些字符串文字类似“`^From:(.*)`”。对大多数程序员，

尤其新入行的程序员来说，有一点难以理解：在构造作为正则表达式的字符串时，他们还需要留意编程语言定义的字符串元字符。

每种语言的字符串文字都规定了自己的元字符，有些语言甚至包含了多种字符串文字，所以不存在普适性的规则，不过背后的概念是一样的。许多语言的字符串文字能够识别转义序列，例如\t、\\和\x2A，在生成字符串对应的数据时，会正确地解释这些记号。与正则表达式相关的最常见的一点就是，在字符串文字中，必须使用两个紧挨在一起的反斜线才能表示正则表达式中的反斜线。例如，为了表示正则表达式中的「\n」，必须在字符串中使用"\\n"。

如果忘了添加反斜线，而只是使用"\n"，在大多数语言中，结果「↵」恰好等于「\n」(译注 2)。不过，事实上，如果正则表达式是宽松排列格式的/x 类型，「↵」被解释为空，「\n」仍然留在正则表达式中，匹配一个空行。忘记这一点的程序员真该打。下面的表 3-4 列出了一些包括\t 和\x2A（2A 是‘*’符号的 ASCII 编码）的例子。表格中的第二对例子展示了忽略字符串文字元字符会导致的意外结果。

表 3-4：关于字符串文字的若干例子

字符串文字	"[\t\x2A]"	"[\\t\\x2A]"	"\t\x2A"	"\\t\\x2A"
字符串的值	‘[⇥*]’	‘[\t\x2A]’	‘⇥*’	‘\t\x2A’
作为正则表达式	「[⇥*]」	「[\t\x2A]」	「⇥*」	「\t\x2A」
能够匹配	星号或制表符	星号或制表符	任意数目的制表符	制表符和之后的星号
在/x 模式下	星号或制表符	星号或制表符	错误	制表符和之后的星号

语言不同，字符串文字也不相同，不过有的差异大到连‘\’都不算元字符。例如，VB.NET 的字符串文字只有一个元字符，就是双引号。下一节介绍了几种常用语言的字符串文字。无论规定如何，我们在使用时都不要忘记考虑“在编程语言的字符串处理结束之后，正则引擎接收到的是什么？”

Java 的字符串

Java 的字符串跟上面提到的很类似，它们由双引号标注，反斜线是元字符。支持常见的字符组合，例如‘\t’（制表符）、‘\n’（换行符）、‘\\’（反斜线本身）。字符串中出现未获支持的反斜线转义序列会出错。

译注 2：此时"\n"是在字符串中识别的，而不是由正则引擎识别。

VB.NET 的字符串

VB.NET中的字符串同样是由双引号标注的，不过它们与Java的字符串有很大差别。VB.NET的字符串只能识别一个元字符：两个连续的双引号，代表字符串中的双引号。例如"he said ""hi""\."的值就是「he said "hi"\.」。

C#的字符串

尽管微软的.NET Framework中所有语言在内部共享同一台正则引擎，但创建正则表达式时它们有各自的规定。我们刚刚看到，Visual Basic的字符串文字非常简单。与之不同，C#语言有两种类型的字符串文字。

C#支持与导论中类似的常见的双引号字符串，只是用 "" 而不是 \" 来表示双引号。不过，C#也支持"原生字符串（verbatim strings）"（译注3），其形式为@"…"。原生字符串不能识别反斜线序列，不过其中也有一个特殊的转义序列：一对双引号表示目标字符串中的一个双引号。也就是说，你可以使用"\\t\\x2A"或者@"\t\x2A"来生成「\t\x2A」。因为这种方式很简单，一般都用@"…"的原生字符串来表示正则表达式。

PHP 的字符串

PHP也提供了两种类型的字符串，不过无一与C#中的相同。在PHP的双引号字符串中可以使用常见的反斜线序列——例如'\n'，但也可以像Perl那样进行变量插值（☞77），还可以使用特殊的序列{…}，把执行花括号内代码的执行结果插入字符串。

PHP的双引号字符串的独特性在于，你可能倾向于在正则表达式中加入多余的反斜线，不过PHP的另一种特性能够缓解这种现象。对Java和C#的字符串文字来说，字符串中如果出现不能明确识别为特殊字符的反斜线序列会导致错误，而在PHP的双引号字符串中，这种序列会原封不动地从字符串中传过来。PHP的字符串能够识别\t，所以你需要用"\\t"来表示「\t」，不过如果使用"\w"，我们仍然得到「\w」，因为\w不属于PHP的字符串能够识别的转义序列。这个额外的特性，虽然有时候很顺手，也增加了PHP双引号字符串的复杂程度，所以PHP提供了更加简单的单引号字符串。

PHP的单引号字符串类似VB.NET字符串和C#的@"…"字符串，都属于"格式整齐的（uncluttered）"字符串，不过稍有不同。在PHP的单引号字符串中，\'表示单引号，\\表示反斜

译注3：verbatim表示"原封不动的，一字不差的"，故此处翻译为"原生"。

线。任何其他字符（包括任何反斜线）都不会被识别为特殊字符，而会被当作字符的值。也就是说，'\t\x2A'创建「\t\x2A」。因为单引号字符串很简单，用它来表示 PHP 的正则表达式非常方便。

PHP 的单引号字符串在第 10 章有详细讲解（☞445）。

Python 的字符串

Python 提供了好几种字符串文字。单引号和双引号都可以用来创建字符串，不过与 PHP 不同的是，这两种方法没有区别。Python 也提供了“三重引用（triple-quoted）”的字符串，也就是'''…'''或者"""…"""，它们可以包含未转义的换行符。这 4 种类型都支持常用的反斜线序列，例如\n，不过和 PHP 一样，它们也会把不能识别的反斜线序列作为纯字符序列来对待。而在 Java 和 C#中，这些序列会被出错。

与 PHP 和 C#一样，Python 也提供了另一种字符串文字，也就是“原字符串（raw string）”。它类似 C#中的@"…"，Python 在以上 4 种表示法前添加'r'来表示纯字符串。例如，r"\t\x2A"表示「\t\x2A」。与其他语言不同的是，在 Python 的原字符串中，所有的反斜线都会保留，即使是用来转义双引号的（所以双引号可以保存在字符串中）也是如此：r"he said \"hi\"\."表示「he said \"hi\"\.」在使用正则表达式时，这并不是一个真正的问题，因为 Python 的正则表达式流派把「\"」识别为「"」，不过如果你喜欢，你可以忽略这些细节，使用这 4 种纯字符串中的任意一种：r'he said "hi"\.'。

Tcl 中的字符串

Tcl 与其他语言都不一样，因为它没有真正的字符串变量。相反，命令行被分解成“单词”，Tcl 的命令把这些单词作为字符串、变量名和正则表达式，或者其他适合的类型。因为命令行被分解成单词，常见的反斜线序列，例如\n，能够识别和转换，而无法识别的反斜线序列则被忽略。如果愿意，你可以在单词两端添加双引号，不过这并不是必须的，除非中间存在空格。

Tcl 同样也有和类似 Python 的纯字符串类似的原字符串类型，不过 Tcl 使用花括号{…}，而不是 r'…'。在花括号之间，除\n 之外的所有内容都会原封不动地保存下来，所以{\t\x2A}表示「\t\x2A」。

在花括号之内，你可以按自己的意愿添加多组括号。非嵌套的括号必须用反斜线转义，不过反斜线会保留在字符串之中。

Perl 的正则表达式文字

至今为止，我们曾看到过的 Perl 的例子中，正则表达式都是以文字方式提供的（"正则表达式文字（regular expression literals)"）。不过，我们也可以用字符串变量提交正则表达式，例如：

```
$str =~ m/(\w+)/;
```

也可以这样：

```
$regex = '(\w+)';
$str =~ $regex;
```

或者是这样

```
$regex = "(\\w+)";
$str =~ $regex;
```

（不过，使用字符串可能会大大降低效率，☞242，348)。

对于以文字方式提交的正则表达式，Perl 会提供一些额外的特性，包括：

- 变量插值（把变量的值写入正则表达式)。
- 通过「\Q…\E」（☞113）支持文字文本。
- 能够（optional）支持\N{*name*}结构，这样就能通过正式的 Unicode 名来指定字符。例如，「\N{INVERTED EXCLAMATION MARK}Hola!」能够匹配'¡Hola!'。

在 Perl 中，正则表达式文字会作为特殊的字符串进行解析。实际上，这些特性在 Perl 双引号字符串中也有提供。必须说明的一点是，这些特性不是由正则引擎提供的。因为 Perl 中使用的绝大多数正则表达式都是作为正则表达式文本的，许多人认为「\Q…\E」属于 Perl 的正则表达式语言，不过如果你用正则表达式从一个配置文件（或者命令行）读入数据，知道哪些特性是由语言的哪些部分提供的就很重要了。

更多细节，请参考第 7 章第 288 页。

字符编码

Character-Encoding Issues

字符编码是一种写明的共识，它规定不同数值的字节应该如何解释。在 ASCII 编码中，值为十进制 110 的字节代表字符'n'，不过在 EBCDIC 编码中代表'>'。为什么会这样？因为这是由不同的人规定的，没有明确的标准判断各种编码的优劣。字节的值是一样的，不一样的是解释。

ASCII 只定义了单个字节能够代表的所有数值的一半，ISO-8859-1 编码（通常称为“Latin-1 编码”）填补了下面的空间，其中增加了读音字符（accented character）和特殊符号，因而能够被更多的语言所使用。对这种编码来说，值为十进制数 234 的字节被解释为 ê，而在 ASCII 中没有定义。

对我们来说，重要的问题在于：如果我们期望使用某种特定编码的数据，程序是否会这样做？例如，如果我们使用 Latin-1 编码中值分别为 234、116、101 和 115 的 4 个字节（表示法语单词“ê tes”），我们期望使用正则表达式「^\w+$」或者「^\b」来匹配。如果程序中的\w 或者\b 能够支持 Latin-1 字符，就可以正常工作，否则不行。

编码的支持程度

编码有许多种，当你需要关注一种具体的编码时，你需要考虑的重要问题包括：

- 程序能够识别这种编码吗？
- 程序如何决定采用哪种编码来处理这些数据？
- 正则表达式对这种编码的支持程度如何？

编码的支持程度包括若干重要的问题：

- 是否能够支持多字节字符？点号和[^x]之类的表达式是匹配单个**字符**，还是单个**字节**？
- \w、\d、\s、\b 之类的元字符，是否能识别编码中的所有字符？例如，虽然 ê 也是一个字符，\w 和\b 能处理吗？
- 程序是否会扩展对字符组的解释？[a-z]能否匹配 ê？
- 不区分大小写的匹配是否能对所有字符有效？例如，ê 和 Ê 是否一样？

有时候事情不像看起来那么简单。例如，java.util.regex 包的\b 能够正确识别 Unicode 中所有与单词相关的字符，\w 则不能（它只能匹配 ASCII 中的字符）。我们会在本章的其他部分看到更多的例子。

Unicode

Unicode

Unicode 究竟是什么，似乎存在许多误解。从最基本的意义上说，Unicode 是一组**字符设定**，或者是从数字和字符之间的逻辑映射的**概念编码**。例如，韩语字符𠵊对应数字 49,333。这个数值，称为一个“**代码点**（code point）”，通常用十六进制来表示，以“U+”开头。49,333

换算成十六进制是 C0B5，所以싵的代码就是 U+C0B5。针对许多字符，Unicode 还定义了一组属性，例如 3 是一个数字，而 É 是与 é 对应的大写字母。

目前，我们还没有谈到这些数值在计算机上是如何编码为数据的。这样的编码方式有许多，包括 UCS-2 编码（所有的字符都占用两个字节），UCS-4 编码（所有字符占用 4 个字节），UTF-16（大部分字符都占用两个字节，有一些字符占用 4 个字节），以及 UTF-8 编码（用 1 到 6 个字节来编码字符）。具体的程序内部到底使用哪种编码通常不需要用户来关心。用户只需要关心如何将外部数据（例如从文件读入的数据）从已知的编码（ASCII、Latin-1、UTF-8 等）转换给具体的程序。支持 Unicode 的程序通常提供了多种编码和解码程序来进行这些转换。

支持 Unicode 的程序中的正则表达式通常支持\u*num* 元序列，用来匹配一个具体的 Unicode 字符（☞117）。这个数值通常是一个 4 位的十六进制数，所以\uC0B5 表示싵。一定要弄清楚的是，\uC0B5 的意思是“匹配编号为 U+C0B5 的 Unicode 字符”，而没有说具体需要比较哪些字节，因为具体的字节是由代表这个 Unicode 代码点的编码方式在内部决定的。如果程序内部使用的是 UTF-8 编码，这个字符就用 3 个字节表示。不过使用支持 Unicode 程序的用户，并不需要关心这个（也有时候需要，例如使用 PHP 的 preg 套件和模式修饰符 u ☞447）。

还有一些你或许需要知道的相关知识……

字符，还是组合字符序列

一般人眼里的“字符”（character）并不都会被 Unicode 或者支持 Unicode 的程序（或者正则引擎）看作一个字符。例如，有人或许认为 à 是一个字符，但是在 Unicode 中，它可能由两个代码点构成，U+0061(a)和钝重音（grave accent）U+0300(`)。Unicode 提供了许多**组合字符**（combining character），用来修饰（结合）一个基本字符。这会给正则引擎带来些麻烦，例如，点号是应该匹配单个代码点呢，还是整个 U+0061 和 U+0300？

在实践中，许多程序似乎把“字符”和“代码点”视为等价，也就是说，点号可以匹配单个的代码点，无论是基本字符还是组合字符。所以，à（U+0061 加上 U+0300）能够由「^..$」匹配，而不是「^.$」。

Perl 和 PCRE（以及 PHP 的 preg 套件）支持\X 元序列，这样点号（“匹配单个字符”）就能够匹配一个结合了组合字符的基本字符。详见第 120 页。

在支持 Unicode 的编辑器中输入正则表达式时，一定要记住组合字符的概念。如果一个带音调的字符，例如 Å，被正则表达式当作‘A’和‘˚’，很可能无法匹配字符串中单个代码点表示的 Å（下一节讨论单代码点的情况）。同样，对正则引擎来说它是两个不同的字符，所以「[…Å…]」在字符组中添加了两个字符，等于「[…A˚…]」。

同样，如果两个代码点的字符——例如 Å——后面跟有一个量词，量词作用的其实是第二个代码点，也就是「A˚+」。

用多个代码点表示同一个字符

从理论上说，Unicode 应该是代码点和字符之间的一一映射（译注 4），不过在许多情况下，一个字符可能有多种表现方式。前一节中我们看到 à 可以表示为 U+0061 加上 U+0300。不过，它也可以用单个代码点 U+00E0。为什么会出现这种情况？是为了保证 Unicode 和 Latin-1 之间转换的简易性。如果我们有需要转换为 Unicode 的 Latin-1 文本，à 可能被转换为 U+00E0。不过，也可以转换为 U+0061 和 U+0300 的组合。通常，这种转换是自动的，用户无法干预，不过 Sun 的 java.util.regex 包提供了一种特殊的匹配符，CANON_EQ，保证能够匹配“在规则中等价（canonically equivalent）”的字符，无论它们在 Unicode 中使用什么存储方式（☞368）。

与此相关的问题是，不同的字符可能无法从外观上区分，如果需要检查生成的文本，这会带来混乱。例如，罗马字母 I（U+0049）可能与 I，也就是希腊字母 Iota（U+0399）混淆。这个字符添加希腊语冒号之后得到 Ï 或者 Ϊ，编码也增加到 4 种（U+00CF；U+03AA；U+0049 U+0308；U+0399 U+0308）。也就是说，如果需要匹配 Ï，你可能需要手动指定这 4 种可能。类似的例子还有许多。

译注 4：即离散数学中的“双射”。

还有许多单个字符看起来不只一个字符。例如 Unicode 定义了一个叫做“SQUARE HZ”（U+3390）的字符。这很像两个普通字符 Hz 的组合（U+0048 U+007A）。

尽管 Hz 之类的特殊字符的用途在目前非常有限，但在将来，它们的应用肯定会增加文本处理程序的复杂性，所以在处理 Unicode 时不应忘记这些问题。用户可能会期望，处理这样的数据时，必须能够处理正常空格（U+0020）和非换行空格（no-break spaces）（U+00A0），或许还需要包括 Unicode 中其他的成打的空白字符之中的任意一个。

Unicode 3.1+和 U+FFFF 之后的代码点

Unicode Version 3.1 诞生于 2001 年中期，增加了 U+FFFF 之后的代码点（之前版本的 Unicode 也支持这些代码点，但是在 Version 3.1 以前，它们都是没有定义的）。例如，代表音乐谱号 C（Clef）的字符对应代码点 U+1D121。之前那些仅支持低于 U+FFFF 字符的程序无法处理这种情况。大多数程序的\u*num* 只能支持最多 4 位十六进制数值。

能够处理这类新字符的程序通常提供了\x{*num*}序列，*num* 可以为任意多位数字（这是为了增强只支持 4 位数字的\u*num* 表示法）。你可以使用\x{1D121}来匹配这类“谱号 C”之类的字符。

Unicode 中的行终止符

Unicode 定义了多个用于表示**行终止符**的字符（以及一个双字符序列），详见表 3-5。

表 3-5：Unicode 行终止符

字符		描述
LF	U+000A	ASCII 换行符
VT	U+000B	ASCII 垂直制表符
FF	U+000C	ASCII 进纸符
CR	U+000D	ASCII 回车
CR/LF	U+000D U+000A	ASCII 回车/换行
NEL	U+0085	Unicode 换行
LS	U+2028	Unicode 行分隔符
PS	U+2029	Unicode 段分隔符

如果行终止符获得了完全的支持，它会影响文本行从文件（在脚本语言中，还包括程序读取的文件）读入的方式。在使用正则表达式时，它们影响点号（☛111），以及「^」、「$」和「\Z」的匹配（☛112）。

正则模式和匹配模式

Regex Modes and Match Modes

许多正则引擎都支持多种不同的模式，它们规定了正则表达式应该如何解释和应用。我们已经看过 Perl 的/x 修饰符（容许自由空格和注释的正则模式☛72）和/i 修饰符（进行不区分大小写匹配的模式☛47）。

在许多流派中，模式可以完全作用于整个表达式，也可以单独应用于某个子表达式。整体应用是通过修饰符或者选项（options）来决定的，例如 Perl 的/i，PHP 的模式修饰符 i（☛446），和 java.util.regex 的 Pattern.CASE_INSENSITIVE 标志（☛99）。如果支持，应用到目标字符串中部分文本的模式是通过一个正则结构来实现的，例如用「(?i)」来开启不区分大小写的匹配，「(?-i)」来停用该匹配。有的流派也支持「(?i:…)」和「(?-i:…)」来启用或者停用对括号内的子表达式进行不区分大小写匹配的功能。

本章后面部分会介绍（☛135）如何在正则表达式中设置这些模式。在本节，我们只看看大多数系统提供的常见模式。

不区分大小写的匹配模式

此模式很常见，它在匹配过程中会忽略字母的大小写，所以「b」可以匹配‘b’和‘B’。此功能也必须依赖于正确的字符编码支持，所以之前我们提到的注意事项对它都适用。

在历史上，不区分大小写的匹配支持一直不太令人满意，被 bug 困扰，好在如今大部分已经修正了。不过，Ruby 不区分大小写的匹配仍然不能处理八进制和十六进制的转义字符。

不区分大小写的匹配存在特殊的与 Unicode 相关的问题(在 Unicode 中称为“粗略匹配(loose matching)”)。简单地说，就是并非所有的 ASCII 字母和数字字符都存在大小写形式，而某些字符在作为单词首字母时会有单独的**标题格式**（title case）。有时候在大写和小写之间并没有明显的一对一映射。常见的例子是希腊字母西格马 Σ，它有两个小写形式 ς 和 σ，在不区分大小写的模式中，这三者应该是等价的。根据我的测试，只有 Perl 和 Java 的 java.util.regex 能够正确处理它们。

另一个问题是，有时候单个字符会对应到一组字符。常见的例子是大写的 ß 是两个字符的组合“SS”。这种情况只有 Perl 能够正确处理。

Unicode 还带来了一些问题。例如单字符 ǰ（U+01F0）没有对应的大写形式的单字符。相反，J̌ 需要使用组合字符（☞107），U+004A 和 U+030C。而 ǰ 和 J̌ 应该在不区分大小写的模式中是等价的。类似的还有一对三的例子。幸好，这些都不是常用字符。

宽松排列和注释模式

此模式会忽略字符组外部的所有空白字符。字符组内部的空白字符仍然有效（java.util.regex 是例外），#符号和换行符之间的内容视为注释。我们已经见过 Perl（☞72）、Java（☞98）和 VB.NET（☞99）中相应的例子。

不过，在 java.util.regex 中，字符组之外的**所有**空白字符并非都会被忽略，而是作为一个“无意义元字符（do-noting metacharacter）”。在理解「\12·3」时，这种区分很重要，因为它表示「3」接在「\12」之后，而不是有些人以为的「\123」。

当然，“空白字符”的定义取决于所采用的字符编码的定义，以及此编码对空白字符的支持程度。大多数程序只能识别 ASCII 的空白字符。

点号通配模式（dot-match-all match mode，也叫“单行模式”）

通常，点号是不能匹配换行符的。最初的 Unix 正则表达式工具是逐行处理的，直到 *sed* 和 *lex* 出现之后，才提出匹配换行符的要求。那时候，人们常用「.*」来匹配“本行中的其他内容（the rest of the line）”，为了保证一致，新的语法不能修改「.*」的定义（注 9）。所以，能够处理多行文本的工具（例如文本编辑器）通常不容许点号匹配换行符。

对现代编程语言来说，点号能够匹配换行符的模式和不能匹配的模式同样有用。这两种模式哪个更方便，取决于具体的情况。许多程序提供了两种方法供正则表达式选择。

这种常规标准也有少数例外的情况。支持 Unicode 的系统，例如在 Sun 的正则表达式包，点号能够匹配未使用此模式时点号不能匹配的所有单字符 Unicode 行终止符（☞109）。在

注 9：Ken Thompson（*ed* 的作者）向我解释说，这样「.*」变得“非常古怪”。

Tcl 的普通模式中，点号能够匹配任何字符，但是在其特殊的“区分换行（newline-sensitive）”和“部分区分换行（partial newline-sensitive）”的匹配模式下，点号和排除型字符组都不能匹配换行符。

不幸的命名

`/s` 修饰符对应的匹配模式第一次出现在 Perl 时，被称为“单行文本模式（single-line mode）”。这个不幸的命名一直是混乱的起源，因为与下一节讨论的“多行文本模式（multiline mode）”比较起来，它似乎与「^」和「$」没有关系。其实“单行文本模式”指的是，点号不受限制，可以匹配任何字符。

增强的行锚点模式（Enhanced line-anchor match mode，也叫“多行文本模式”）

增强的行锚点模式会影响到行锚点「^」和「$」的匹配。通常情况下，锚点「^」不能匹配字符串内部的换行符，而只能匹配目标字符串的起始位置。但是在此增强模式下，它能够匹配字符串中内嵌的文本行的开头位置。前一章出现了这样的例子（☞69），当时我们用 Perl 开发把文本内容转换为超文本内容程序。其中，所有的文本保存在一个字符串中，所以我们可以通过查找-替换功能用 `s/^$/<p>/mg` 来把“…tags. ␤ ␤It's…”转换为“…tags. ␤ **<p>**␤It's…”。该替换把空“行”替换为段落 tag。

「$」也是这样，尽管「$」在正常情况下的匹配的基本规则比较难理解（☞129）。不过，就本节来说，我们只需要记住，「$」可以匹配字符串内部的换行符，就足够了。

支持此模式的程序通常还提供了「\A」和「\Z」，它们的作用与普通的「^」和「$」一样，只是在此模式下它们的意义**不会**发生变化。也就是说「\A」和「\Z」永远不会匹配字符串内部的换行符。有些实现方式中，「$」和「\Z」能够匹配字符串内部的换行符，不过它们通常会提供「\z」，唯一匹配整个字符串的结尾位置。详见 129 页。

对点号来说，常用标准有一些例外。在 GNU Emacs 之类的文本编辑器中，行锚点通常能够匹配字符串中的换行符，因为在编辑器中这样非常有意义。另一方面，*lex* 的「$」只能匹配换行符之前的位置（其中「^」的意义与常见的一样）。

此模式下，在 Sun 的 `java.util.regex` 之类支持 Unicode 的系统中，行锚点能够匹配任何一种行终止符（☞109）。Ruby 的行锚点在正常情况下**能够**匹配字符串中的换行符，Python 的「\Z」类似「\z」，而不是普通的「$」。

长期以来，这种模式被称为“多行模式（multiline mode）”。尽管它与“单行模式”没有什么关系，但看名字总容易觉得二者有关联。后者修改的是点号的匹配规则，前者修改的是「^」和「$」的匹配规则。另一方面，它们从不同的思路处理换行符。第一个修改了点号处理换行符的方式，从“需要特殊处理”变为“不需要特殊处理”；第二个的做法则相反，改变了「^」和「$」匹配换行符的方式，从“不需要特殊处理”变为“需要特殊处理”（注 10）。

文字文本模式

“文字文本（literal text）”模式几乎不能识别任何正则表达式元字符。例如，文字文本模式下「[a-z]*」匹配字符串“[a-z]*”。完整的文字搜索（literal search）等于简单的字符串搜索（“搜索这个字符串”，而不是“搜索这个正则表达式”），支持正则表达式的程序通常也提供了普通的字符串搜索功能。正则表达式的文字文本模式之所以更有趣，是因为它可以只作用于正则表达式的一部分，举例来说，PCRE（因此也包括 PHP）的正则表达式和 Perl 的正则表达式文本提供了特殊的序列\Q…\E，其中内容的元字符全部被忽略（当然，不包括\E）。

常用的元字符和特性

Common Metacharacters and Features

本章的其他部分——剩下大约 30 页的内容简要介绍下一页列出的常见正则表达式元字符和概念。这里的介绍并不是全面彻底的，不过也没有任何一种正则工具涉及其中的所有内容。

从某种意义上说，这一节只是前两章内容的总结，但同时也是为本章介绍更全面更深刻的知识做准备。读者第一次接触时，只需略读本章就可以继续阅读下面各章，以后在需要的时候可以随时回过头来查阅细节。

有的工具添加了大量的新功能，也可能毫无根据地改变某些通用表示法，以满足它们的特殊要求。尽管我有时会提到这些特殊的工具，但不会花太多的笔墨在工具的细节问题上。相反，在这一节我只希望介绍常见的元字符及其作用，以及与此相关的一些问题。我希望读者能够参考自己擅长的工具提供的使用手册。

注 10：正常情况下，Tcl 的点号能够匹配所有字符，所以从这个意义上说它比其他任何语言都要直接易懂。在 Tcl 的正则表达式中，换行符并不需要特殊处理（对点号和行锚点来说都是如此），但是如果明确指定了匹配模式，情况就不一样了。不过，因为其他系统通常以其他方式来做到这一点，那些习惯其他方式的用户可能会感到迷惑。

本节介绍的结构

字符表示法

☞115 字符缩略表示法：\n、\t、\a、\b、\e、\f、\r、\v、…
☞116 八进制转义：*num*
☞117 十六进制/Unicode 转义：\x*num*、\x{*num*}、\u*num*、\U*num*，…
☞117 控制字符：\c*char*

字符组及相关结构

☞118 普通字符组：[a-z]和[^a-z]
☞119 几乎能匹配任何字符的元字符：点号
☞120 单个字节：\C
☞120 Unicode 组合字符序列：\X
☞120 字符组缩略表示法：\w、\d、\s、\W、\D、\S
☞121 Unicode 属性、区块和分类：\p{*Prop*}、\P{*Prop*}
☞125 字符组运算符：[[a-z]&&[^aeiou]]
☞127 POSIX “字符组”方括号表示法：[[:alpha:]]
☞128 POSIX “collating 序列”方括号表示法：[[.span-ll.]]
☞128 POSIX “字符等价类”方括号表示法：[[=n=]]
☞128 Emacs 语法类

锚点及其他“零长度断言”

☞129 行/字符串起点：^、\A
☞129 行/字符串终点：$、\Z、\z
☞130 本次匹配的开始位置（或者上次匹配的结束位置）：\G
☞133 单词分界符：\b、\B、\<、\>，…
☞133 顺序环视 (?=…)、(?!…)；逆序环视 (?<=…)、(?<!…)

注释和模式修饰词

☞135 模式修饰词：(?*modifier*)，例如(?i)或(?-i)
☞135 模式作用范围：(?*modifier*:…)，例如(?i:…)
☞136 注释：(?#…) 和 #…
☞136 文字文本范围：\Q…\E

分组、捕获、条件判断和控制

☞137 捕获/分组括号：(…)、\1、\2，…
☞137 仅用于分组的括号：(?:…)
☞138 命名捕获：(?<*Name*>…)
☞139 固化分组：(?>…)
☞139 多选结构：…|…|…
☞140 条件判断：(?*if then*|*else*)
☞141 匹配优先量词：*、+、?、{*num*,*num*}
☞141 忽略优先量词：*?、+?、??、{*num*,*num*}?
☞142 占有优先量词：*+、++、?+、{*num*,*num*}+

字符表示法

Character Representations

这一组元字符能够以清晰美观的方式匹配其他方式中很难描述的某些字符。

字符缩略表示法

许多工具软件提供了表示某些控制字符的元字符，其中有一些在所有机器上都是不变的，但也有些是很难输入或观察的：

- `\a` **警报**（例如，在“打印”时扬声器发声）。通常对应 ASCII 中的<BEL>字符，八进制编码 007。
- `\b` **退格** 通常对应 ASCII 中的<BS>字符，八进制编码 010。（在许多流派中，「\b」只有在字符组内部才表示这样的意义，否则代表单词分界符☞133）。
- `\e` **Escape 字符** 通常对应 ASCII 中的<ESC>字符，八进制编码 033。
- `\f` **进纸符** 通常对应 ASCII 中的<FF>字符，八进制编码 014。
- `\n` **换行符** 出现在几乎所有平台（包括 Unix 和 DOS/Windows）上，通常对应 ASCII 的<LF>字符，八进制编码 012。在 MacOS 中通常对应 ASCII 的<CR>字符，十进制编码 015。在 Java 或任意一种.NET 语言中，不论采用什么平台，都对应 ASCII<LF>字符。
- `\r` **回车** 通常对应 ASCII 的<CR>字符。在 MacOS 中，对应到 ASCII 的<LF>字符。在 Java 或任意一种.NET 语言中，不论采用什么平台，都对应到 ASCII 的<CR>字符。
- `\t` **水平制表符** 对应 ASCII 的<HT>字符，八进制编码 011。
- `\v` **垂直制表符** 对应 ASCII 的<VT>字符，八进制编码 013。

表 3-6 列出了几种常用的工具及它们提供的某些字符缩略表示法。之前已经说过，某些语言在支持字符串文字时已经提供了同样的字符缩略表示法。请不要忘记那一节（☞101），因为它涉及某些相关的陷阱。

会根据机器变化的字符？

从该表可以看出，在许多工具中，\n 和\r 的意义是随操作系统的变化而变化的（注 11），所以在使用时应格外小心。如果你需要在程序可能运行的所有平台上都能通用的“换行符”，请使用\n。如果需要一个对应特殊值的字符，例如 HTTP 协议定义的分隔符，请使用\012

注 11：如果工具软件本身是用 C 或者 C++写的，将其中的正则表达式反斜线转义转换为 C 的反斜线转义，结果取决于编译器。在现实中，在特定平台的各种编译器对换行符的支持是有统一标准的，所以我们可以认为这个问题只与操作系统相关。不过，因为只有\r 和\n 的意义会根据操作系统的变化（在一定程度上，也与时间有关），可以认为其他的字符在所有系统上都是统一的。

表 3-6：几款工具软件及它们提供的元字符简写法

	单词分界符 `\b`	退格字符 `\b`	警报 `\a`	ASCII Escape `\e`	进纸符 `\f`	换行符 `\n`	回车符 `\r`	制表符 `\t`	垂直制表符 `\v`
程序		字符简写							
Python	✓	✓$_{C}$	✓		✓	✓	✓	✓	✓
Tcl	等于\y	✓	✓	✓	✓	✓	✓	✓	✓
Perl	✓	✓$_{C}$	✓	✓	✓	✓	✓	✓	
Java	✓$_{X}$	✓̸$_{X}$	✓	✓	✓$_{SR}$	✓$_{SR}$	✓$_{SR}$	✓$_{SR}$	✓
GNU awk		✓	✓		✓	✓	✓	✓	✓
GNU sed	✓					✓			
GNU Emacs	✓	✓̸$_{S}$	✓̸$_{S}$	✓̸$_{S}$	✓̸$_{S}$	✓̸$_{S}$	✓̸$_{S}$	✓̸$_{S}$	✓̸$_{S}$
.NET	✓	✓$_{C}$	✓	✓	✓	✓	✓	✓	✓
PHP（preg 套件）	✓	✓$_{C}$	✓	✓	✓	✓	✓	✓	
MySQL									
GNU grep/egrep	✓								
flex		✓	✓		✓	✓	✓	✓	✓
Ruby	✓	✓$_{C}$	✓	✓	✓	✓	✓	✓	✓

✓支持；✓$_{C}$ 只在字符组内部支持

✓$_{SR}$ 支持（字符串文字也支持）

✓$_{X}$ 支持（但在字符串文字中，同样的序列有不同的意义）

✓̸$_{X}$ 不支持（但在字符串文字中，同样的序列有不同的意义）

✓̸$_{S}$ 不支持（但字符串文字支持）

本表格假设在每种程序中使用的都是最适合正则表达式的字符串类型

版本信息请参考第 91 页

之类标准规定的字符（`\012` 是 ASCII 中的换行符的八进制编码）。如果你希望匹配 DOS 中的行终结字符，请使用「`\015\012`」。如果希望同时匹配 DOS 或 Unix 的换行字符，请使用「`\015?\012`」（它们通常是匹配行尾的字符，如果希望匹配行的开头位置或结尾位置，请使用行锚点☞129）。

八进制转义*num*

支持八进制（以 8 为基数）转义的实现方式通常容许以 2 到 3 位数字表示该值所代表的字节或字符。例如，「`\015\012`」表示 ASCII 的 CR/LF 序列。八进制转义可以很方便地在正则表达式中插入平时难以输入的字符。例如，在 Perl 中，我们可以使用「`\e`」作为 ASCII 的转义字符，但是在 awk 中不行。

因为 awk 支持八进制转义，我们可以直接使用 ASCII 代码来表示 escape 字符：「\033」。

下一页的表 3-7 列出了部分工具支持的八进制转义。

有些实现方式很特殊，在其中「\0」能够匹配字节 NUL。有的支持一位数字的八进制转义，不过如果同时支持「\1」之类的反向引用，就不会提供这种功能。如果两者发生冲突，则反向引用一般要优先于八进制转义。有的容许出现 4 位数字的八进制转义，不过通常会要求任何八进制转义都必须以 0 开头（例如 java.util.regex）。

你可能会想，如果遇到\565 之类超出范围的转义数值（8 位的八进制数值范围从\000 到\377）会发生什么。大约一半的实现方式将其视为多余一个字节的值（如果支持 Unicode，则可能是 Unicode 字符），而其他实现方式会将其截断为一个字节。一般来说，最好的办法还是不要使用超过\377 的八进制转义。

十六进制及 Unicode 转义：\x*num*、\x{*num*}、\u*num*、\U*num*

除了八进制转义之外，许多工具软件也支持十六进制转义（以 16 为基数），以\x、\u 或者是\U 开头。如果支持\x，则「\x0D\x0A」匹配 CR/LF 序列。表 3-7 列出了部分工具软件支持的十六进制转义。

除了知道采用的是哪种转义之外，你可能还希望知道各种转义能识别多少位的转义值，以及是否能够（或者必须）在数字两端使用花括号。对此，表 3-7 同样作了说明。

控制字符：\c*char*

许多流派中可以用「\c*char*」来匹配编码值小于 32 的**控制字符**（有些能支持更大的值）。例如，「\cH」匹配一个 Control-H 字符，也就是 ASCII 中的退格符，而「\cJ」匹配 ASCII 的换行符（通常使用「\n」，不过有时也使用「\r」，这取决于具体的平台☞115）。

支持此结构的系统在细节上有所不同。与这个例子一样，通常使用大写英文字母是不会有问题的。在大多数实现方式中，你也可以使用小写字母，不过也有的软件不支持它们，例如 Sun 的 Java regex Package。流派不同，对字母和数字之外的字符的处理是非常不同的，所以我推荐在使用\c 时只使用大写字母。

相关提示：GNU Emacs 支持此功能，但它使用的元序列非常奇特「?\^*char*」（例如：「?\^H」匹配 ASCII 编码中的退格字符）。

表 3-7：部分工具软件及它们的正则表达式支持的八进制和十六进制转义

	反向引用	八进制转义	十六进制转义
Python	✓	\0、\07、\377	\xFF
Tcl	✓	\0、\77、\377	\x…\uFFFF；\UFFFFFFFF
Perl	✓	\0、\77、\377	\xF；\xFF；\x{…}
Java	✓	\07、\77、\0377	\xFF；\uFFFF
GNU awk		\7、\77、\377	\x…
GNU sed	✓		
GNU Emacs	✓		
.NET	✓	\0、\77、\377	\xFF；\uFFFF
PHP（preg 套件）	✓	\0、\77、\377	\xF，\xFF，\x{…}
MySQL			
GNU egrep	✓		
GEU grep	✓		
flex		\7、\77、\377	\xF；\xFF
Ruby	✓	\7、\77、\377	\xF；\xFF

\0——「\0」匹配字节 NUL，而其他一位数字的八进制转义是不支持的。
\7,\77—— 一位和两位八进制转义都支持
\07——支持开头为 0 的两位八进制转义
\077 ——支持开头为 0 的 3 位八进制转义
\377 ——支持不超过\377 的 3 位八进制转义
\0377 ——支持不超过\0377 的 4 位八进制转义
\777 ——支持不超过\777 的 3 位八进制转义
\x… ——容许出现任意多位数字
\x{…} ——\x{…}容许出现任意多位数字
\xF、\xFF ——以\x 开头，容许出现一到两位十六进制转义
\uFFFF ——以\u 开头的 4 位十六进制转义
\UFFFF ——以\U 开头的 4 位十六进制转义
\UFFFFFFFF ——以\U 开头的 8 位十六进制数字（参考第 91 页的版本信息）

字符组及相关结构

Character Classes and Class-Like Constructs

现在许多流派中，都有多种方法在正则表达式的某个位置指定一组字符，不过最通行的方法还是使用普通字符组。

普通字符组 ：[a-z]和[^a-z]

我们已经介绍过字符组的基本概念，不过我还是要强调，元字符的规定在字符组内外是有差别的。例如，在字符组内部「*」永远都不是元字符，而「-」通常都是元字符。有些元序列，例如「\b」，在字符组内外的意义是不一样的（☞116）。

在大多数系统中，字符组内部的顺序环视是无关紧要的，而且使用范围表示法而不是列出范围内的所有字符并不会影响执行速度（例如，[0-9]与[908176354]是一样的)。相反，某些实现方式不能完全优化字符组（比如 Sun 提供的 Java regex package)，所以最好是使用范围表示法，因为如果有差别，这种表示法的速度会快一些。

字符组通常表示**肯定断言**（positive assertion)。也就是说，它们必须匹配一个字符。排除型字符组仍然需要匹配一个字符，只是它**没有**在字符组中列出而已。把排除型字符组理解为"匹配未列出字符的字符组"或许更容易一些（请务必阅读下一节中关于点号和排除型字符组的警告)。「[^LMNOP]」通常等价于「[\x00-kQ-\xFF]」，即使在规定严格的 8 位系统中，这仍然成立，但是在 Unicode 之类字符的值可能大于 255（\xFF）的系统中，排除型字符组「[^LMNOP]」可能包括成千上万个字符——只是不包含 L、M、N、O 和 P。

请务必理解使用范围表示法的基本字符组。例如，用「[a-Z]」匹配字母就很可能存在遗漏，而且在任何情况下显然都不是"所有字母"。而[a-zA-Z]则能匹配所有字母，至少对于 ASCII 编码来说是这样的（请参考"Unicode 属性"中的\p{L}☞121)。当然，在处理二进制数据时，字符组中的'\x80-\xFF'范围表示法完全适用。

几乎能匹配任何字符的元字符：点号

在某些工具软件中，点号用来缩略表示可以匹配任何字符的字符组，而在其他工具中，点号能匹配**除了换行符之外的**任何字符。这差别很细微，但如果所用的工具能够处理包含多个逻辑行的目标文本（或者是文本编辑器中的多个逻辑行的文本)，它就非常重要。关于点号，需要注意的有：

- 在 Sun 的 Java regex package 之类的支持 Unicode 的系统中，点号不能匹配 Unicode 的行终结符（☞109)。
- 匹配模式（☞111）会改变点号的匹配规则。
- POSIX 规定，点号不能匹配 NUL（值为 0 的字符)，尽管大多数脚本语言容许文本中出现 NULL（而且可以用点号来匹配)。

点号，还是排除型字符组

如果所使用的工具能够在多行文本中进行搜索，请务必注意点号，它在通常情况下不能匹配换行符，而排除型字符组「[^"]」通常都可以。如果把「".*"」替换为「"[^"]*"」，可能会带来意想不到的效果。点号的匹配规定一般可以通过变换匹配模式来更改——请参考第 111 页的"点号通配模式"。

单个字节

Perl 和 PCRE（也包括 PHP）支持用\C 匹配单个**字节**，即使该字节位于某个多字节编码的字符之中（相反，其他功能都是基于**字符**的）。这个功能很危险，如果运用不当，可能会导致内部错误，所以只有在清楚自己所作所为的情况下，才能使用它。我找不到恰当运用的例子，所以下文不再提及。

Unicode 组合字符序列：\X

Perl 和 PHP 支持用\X 缩略表示「\P{M}\p{M}*」，它可以视为点号的扩展。它匹配一个**基本字符**（除\p{M}之外的任何字符），之后可能有任意数目的**组合字符**（除\p{M}之外）。

之前已经介绍过（☞107），Unicode 体系包括基本字符和组合字符，二者可以合成“看起来”的单个字符，例如 à（‘a’的编码是 U+0061，读音符号‘`’的编码是 U+0300）。有的字符可能包含不止一个的组合字符。例如，‘ç̆’就包括‘c’，然后是变音符‘¸’，最后是短音符‘˘’（Unicode 编码分别是 U+0063、U+0327 和 U+0306）。

如果希望匹配“francais”或者“français”，仅仅使用「fran.ais」或者「fran[cÇ]ais」还不够保险，因为此方法假设‘ç’用单个 Unicode 代码点 U+00C7 表示，而不是‘c’加上变音符（U+0063 加上 U+0327）。如果需要专门处理，可以使用「fran(c,?|Ç)ais」，不过在这里，用「fran\Xais」取代「fran.ais」是个好办法。

除了能够匹配结尾的组合字符之外，\X 与点号还有两个差别。其一是，\X 始终能匹配换行符和其他 Unicode 行终结符（☞109），而点号只有在点号通配模式（☞111），或者工具软件提供的其他匹配模式下才可以。另一点是，点号通配模式下的点号无论什么情况下都能匹配任何字符，而「\X」不能匹配以组合字符开头的字符。

字符组简记法：\w、\d、\s、\W、\D、\S

通常支持的简记法有：

\d **数字** 等价于「[0-9]」，如果工具软件支持 Unicode，能匹配所有的 Unicode 数字。

\D **非数字字符** 等价于「[^\d]」。

\w **单词中的字符** 一般等价于「[a-zA-Z0-9_]」。某些工具软件中「\w」不能匹配下画线，而另一些工具软件的「\w」则能支持当前 locale（☞87）中的所有数字和字符。如果支持 Unicode，「\w」通常能表示所有数字和字符，而在 java.util.regex 和 PCRE（也包括 PHP）中，「\w」严格等价于「[a-zA-Z0-9_]」。

\W **非单词字符** 等价于「[^\w]」。

\s **空白字符** 在支持ASCII的系统中,它通常等价于「[·\f\n\r\t\v]」。在支持Unicode的系统中,有时包含Unicode的"换行"控制字符U+0085,有时包含"空白(whitespace)"属性\p{Z}(参见下一节的介绍)。

\S **非空白字符** 等价于「[^\s]」。

87页已经介绍过,POSIX的locale设定会影响这些简记符号的含义(尤其是\w)。支持Unicode的程序中,\w通常能匹配更多的字符,例如\p{L}(下一节介绍)和下画线。

Unicode属性、字母表和区块:\p{*Prop*}、\P{*Prop*}

表面上看,Unicode只是一套字符映射规则(☞106),其实Unicode标准远远不止这些。它还定义了每个字符的性质(qualities),例如"这个字符是小写字母","这个字符是从右往左看的","这个字符是标记字符(mark),它必须与其他字符一同使用"等等。

不同的正则表达式系统对这些属性的支持也不相同,但是许多支持Unicode的程序能够通过「\p{*quality*}」和「\P{*quality*}」支持其中的一部分。比如「\p{L}」就是个简单的例子,这里'L'的意思是"字母(letter)"(相对于数字number、标点punctuation和口音accent,之类)。'L'是一种**普通属性**(general property,也称为**分类** category)。我们马上会了解到,可以用「\p{…}」和「\P{…}」来测试其他"属性",当然支持最广泛的还是常见的属性。

常见的属性请见表3-8。每个字符(实际上是代码点,包括那些目前没有对应字符的代码点)都可以用一个普通属性匹配。普通属性的名字是单个字符(例如'L'表示字母letter,'S'表示符号symbol,等等),但是某些系统中可以用多个字母表述属性(例如'Letter'和'Symbol'),比如Perl就支持这样。

在某些系统中,单字母属性名可能不需要花括号(例如,用\pL而不是\p{L})。有的系统可能要求(或者是容许)使用'In'或'Is'前缀(例如\p{IsL})。讲解扩展属性(additional qualities)时,我们会见到要求使用Is/In前缀的例子(注12)。

按照表3-9,每个用单字符表示的普通属性可以进一步分为多个双字母表示的子属性(sub-property),例如"字母(letter)"又可以分为"小写字母"、"大写字母"、"标题首字

注12:我们会看到(见第125页的表格),所有的Is/In前缀都有点多余。老版本的Unicode推荐一种做法,而更早的实现推荐另一种。在Perl 5.8的开发过程中,我与负责简化Perl的小组一起工作。目前Perl对此的规定是:如非指定要使用某个Unicode区块,就不应该使用'Is'或者'In',否则就必须使用'In'。

表 3-8：基本的 Unicode 属性分类

分 类	等价表示法及描述
\p{**L**}	\p{**Letter**}——字母
\p{**M**}	\p{**Mark**}——不能单独出现，而必须与其他基本字符一起出现（重音符号、包围框，等等）的字符
\p{**Z**}	\p{**Separator**}——用于表示分隔，但本身不可见的字符（各种空白字符）
\p{**S**}	\p{**Symbol**}——各种图形符号（Dingdats）和字母符号
\p{**N**}	\p{**Number**}——任何数字字符
\p{**P**}	\p{**Punctuation**}——标点字符
\p{**C**}	\p{**Other**}——匹配其他任何字符（很少用于正常字符）

母（titlecase letter）"、"修饰符字母"和"其他字母"。每个代码点能且只能属于一种子属性。

要补充的是，某些实现方式支持特殊的复合子属性，例如用\p{L&}表示"分大小写的（cased）"字母，也就是说「[\p{Lu}\p{Ll}\p{Lt}]」。

表 3-9 还给出了某些实现方式支持的属性名的全称（例如，"Lowercase_Letter"，而不是"Ll"）。按照 Unicode 的标准，各种形式都应该能够接受（例如'LowercaseLetter'、'LOWERCASE_ LETTER'、'Lowercase·Letter'、'lowercase-letter'），不过，为了保持一致，我推荐使用表 3-9 中的形式）。

字母表（*Scripts*）有的系统能够按照**字母表**（书写系统 writing system）的名字以「\p{…}」来匹配。例如，用\p{Hebrew}匹配希伯来文独有的字符（但不包含其他书写系统中常见的字符，例如空格和标点）。

某些字母表是基于语言的（例如印度古哈拉地语、泰国语、切罗基语，等等）。有的覆盖了多种语言（例如拉丁文、西里尔文），还有些语言包含多种字母表，例如日语的字符就来自平假名、片假名、汉语和拉丁语。请读者参考自己系统的文档获取完整的信息。

字母表不会包含特定的书写系统中的所有字符，而只包含独属于（或者几乎独属于）此书写系统中的字符。常见的字符，例如空格和标点不属于任何字母表，而是属于通用的 IsCommon 伪字母表(pseudo-script)，用「\p{IsCommon}」匹配。还有一个伪字母表 Inherited，它包括从其所属的字母表中基本字符继承而来的组合字符。

表 3-9：基本的 Unicode 子属性

属　性	等价表示法及说明
\p{Ll}	\p{Lowercase_Letter}——小写字母。
\p{Lu}	\p{Uppercase_Letter}——大写字母。
\p{Lt}	\p{Titlecase_Letter}——出现在单词开头的字母（例如，字符 Dž是小写字母 dž和大写字母 DŽ的首字母形式）。
\p{L&}	\p{Ll}、\p{Lu}、\p{Lt}并集的简记法。
\p{Lm}	\p{Modifier_Letter}——少数形似字母的，有特殊用途的字符。
\p{Lo}	\p{Other_Letter}——没有大小写形式，也不属于修饰符的字母，包括希伯来语、阿拉伯语、孟加拉语、泰国语、日语中的字母。
\p{Mn}	\p{Non_Spacing_Mark}——用于修饰其他字符的“字符（Characters）”，例如重音符号、变音符号、某些“元音记号”和语调标记。
\p{Mc}	\p{Spacing_Combining_Mark}——会占据一定宽度的修饰字符（各种语言中的大多数“元音记号”，这些语言包括孟加拉语、印度古哈拉地语、泰米尔语、泰卢固语、埃纳德语、马来语、僧伽罗语、缅甸语和高棉语）。
\p{Me}	\p{Encolsing_Mark}——可以围住其他字符的标记，例如圆圈、方框、钻石型等。
\p{Zs}	\p{Space_Separator}——各种空白字符，例如空格符、不间断空格（non-break space），以及各种固定宽度的空白字符。
\p{Zl}	\p{Line_Separator}——LINE SEPARATOR 字符（U+2028）。
\p{Zp}	\p{Paragraph_Separator}——PARAGRAPH SEPARATOR 字符（U+2029）。
\p{Sm}	\p{Math_Symbol}——数学符号、+、÷、表示分数的横线。
\p{Sc}	\p{Currency_Symbol}——货币符号、$、¢、¥、…。
\p{Sk}	\p{Modifier_Symbol}——大多数版本中它表示组合字符，但是作为功能完整的字符，它们有自己的意义。
\p{So}	\p{Other_Symbol}——各种印刷符号、框图符号、盲文符号，以及非字母形式的中文字符，等等。
\p{Nd}	\p{Decimal_Digit_Number}——各种字母表中从 0 到 9 的数字（不包括中文、日文和韩文）。
\p{Nl}	\p{Letter_Number}——几乎所有的罗马数字。
\p{No}	\p{Other_Number}——作为加密符号（superscripts）和记号的数字，非阿拉伯数字的数字表示字符（不包括中文、日文、韩文中的字符）。
\p{Pd}	\p{Dash_Punctuation}——各种格式的连字符（hyphen）和短划线（dash）。
\p{Ps}	\p{Open_Punctuation}——(、︽和《等字符。
\p{Pe}	\p{Close_Punctuation}——)、︾和》等字符。
\p{Pi}	\p{Initial_Punctuation}——«、“、<等字符。
\p{Pf}	\p{Final_Punctuation}——»、’、>等字符。
\p{Pc}	\p{Connector_Punctuation}——少数有特殊语法含义的标点，如下画线。
\p{Po}	\p{Other_Punctuation}——用于表示其他所有标点字符：!、&、.、:、︙等。
\p{Cc}	\p{Control}——ASCII 和 Latin-1 编码中的控制字符（TAB、LF、CR 等）。
\p{Cf}	\p{Format}——用于表示格式的不可见字符。
\p{Co}	\p{Private_Use}——分配与私人用途的代码点（例如公司的 logo）。
\p{Cn}	\p{Unassigned}——目前尚未分配字符的代码点。

区块（*Block*）。区块类似（但是比不上）字母表，区块表示Unicode字符映射表中一定范围内的代码点。例如，Tibetan区块表是从U+0F00到U+0FFF的256个代码点。其中的字符，在Perl和java.util.regex中可以用\p{InTibetan}来匹配，在.NET中可以用\p{IsTibetan}来匹配（细节见后文）。

区块有许多种，包括对应大多数书写系统的区块（希伯来、泰米尔、基本拉丁语、西里尔文等等），以及特殊的字符组型（货币、箭头、文本框、印刷符号等）。

Tibetan是一个典型的区块，因为其中的所有字符都是按照西藏文定义的，此区块之外不存在专属于藏语的字符。不过，区块仍然不如字母表，原因如下：

- 区块可能包含未赋值的代码点。例如，Tibetan区块中大约有25%的代码点没有分配字符。
- 并不是看起来与区块相关的所有字符都在区块内部。例如，在Currency区块中就没有通用的货币符号'¤'，也没有常见的$、¢、£、€和¥（幸好，这时候我们可以用currency属性\p{Sc}）。
- 区块通常包含不相关的字符。例如¥（表示"元"）属于Latin_1_Supplement区块。
- 属于某个**字母表**的字符可能同时包含于多个**区块**。例如，希腊字符同时出现在Greek和Greek_Extended区块中。

对区块的支持比对字母表的支持更普遍。不过这两者很容易混淆，因为在命名上存在许多重叠（例如，Unicode同时提供了Tibetan字母表和Tibetan区块）。

此外，按照下页的表3-10所示，这些命名本身也没有统一的标准。在Perl和java.util.regex中，Tibetan区块表示为「\p{InTibetan}」，但是在.NET中又表示为\p{**Is**Tibetan}（更糟糕的是，在Perl中这是Tibetan**字母表**的另一种表示法）。

其他属性（Other properties/qualities）上面介绍的知识并不是通用的。表3-10详细介绍了它们的适用情况。

要补充的是，Unicode还定义了许多能够通过「\p{…}」结构访问的属性，其中包括字符的书写顺序环视（从左至右还是从右至左，等等）、与字符相关的元音，以及其他属性。有些实现方式还容许用户根据需要临时创建属性。请参考具体的程序提供的文档了解细节。

表 3-10：属性/字母表/区块的支持情况

特　性	Perl	Java	.NET	PHP/PCRE
✓基本属性，例如\p{L}	✓	✓	✓	✓
✓基本属性缩略表示法，例如\pL	✓	✓		✓
基本属性缩略表示法，例如\p{IsL}	✓	✓		
✓基本属性的全名，例如\p{Letter}	✓			
✓复合属性，例如\p{L&}	✓			✓
✓字母表，例如\p{Greek}	✓			✓
字母表全名，例如\p{IsGreek}	✓			
✓区块，例如\p{Cyrillic}	如果没有字母	✓		
✓区块全名，例如\p{InCyrillic}	✓	✓		
区块全名，例如\p{IsCyrillic}			✓	
✓排除功能，例如 P{…}	✓	✓	✓	✓
排除功能，例如\p{^…}	✓			✓
✓\p{Any}	✓	等于\p{all}		✓
✓\p{Assigned}	✓	等于\P{Cn}	等于\P{Cn}	等于\P{Cn}
✓\p{Unassigned}	✓	等于\p{Cn}	等于\p{Cn}	等于\p{Cn}

以✓开头的行是新实现方式中推荐的用法（请参考第 91 页的版本信息）

简单的字符组减法：[[a-z]-[aeiou]]

.NET 提供的字符组“减法”容许我们在字符组中进行减法运算。例如，「[[a-z]-[aeiou]]」匹配的字符就是「[a-z]」能够匹配字符的减去「[aeiou]」能够匹配的字符，也就是 ASCII 编码中小写的非元音字母。

另一个例子是「[\p{P}-[\p{Ps}\p{Pe}]]」，它能够匹配\p{P}中除「[\p{Ps}\p{Pe}]」之外的字符，也就是说，它能匹配除了» 和«之类成对的符号之外的所有标点符号。

完整的字符组集合运算：[[a-z] && [^aeiou]]

Sun 的 Java regex package 中的字符组能够进行完整的集合运算（并、减、交）。它的语法有别于前一节中简单的字符组减法(尤其是，在 Java 中匹配小写非元音字母的字符组[[a-z]&&[^aeiou]])。在详细介绍减法之前，我们先来看两个简单的集合运算：OR 和 AND。

OR 容许用户以字符组方式在字符组中添加字符：[abcxyz]也可以表示为[[abc][xyz]]、[abc[xyz]]或 [[abc]xyz]等等。OR 用来把多个集合合并为新的集合。从概念上说，它

有点像多种语言提供的"按位或"运算符：'|'或是'or'。在字符组中，OR只不过是一种简记法，尽管包括**排除型字符组**在某些情况下更方便。

AND对两个集合进行概念上的"与"运算，只保留同时属于两个字符组的字符。它的写法是在两个字符组中添加特殊的字符组元字符&&。例如[\p{InThai}&&\P{Cn}]，它通过对\p{InThai}和\P{Cn}进行交运算（只保留同时属于两个集合的字符），匹配Thai区块中所有**已经赋值**的代码点。\P{…}中的'P'是大写，匹配不具备此属性的字符，所以\P{Cn}匹配的就是除**未赋值**的代码点之外的代码点，也就是**已经赋值**的代码点（要是Sun能够识别已赋值属性（Assigned quality），就可以用\p{Assigned}替换\P{Cn}）。

请不要混淆OR和AND。它们的含义取决于用户的看法。例如[[this][that]]读作"[this]或者[that]匹配的字符"，其实它的真正意思是"[this]和[that]能够匹配的所有字符"，这只是对同一个问题的两种看法。

相比之下，AND要清楚一些：[\p{InThai}&&\P{Cn}]读作"只匹配在\p{InThai}和\P{Cn}中出现的字符"，尽管它有时候也读作"匹配属于\p{InThai}和\P{Cn}的**交集**中的字符。"

看法的不同可能会造成混乱：我叫做OR和AND的运算，某些人可能叫做AND和INTERSECTION。

以集合运算符进行字符组的减法 \P{Cn}可以写作[^\p{Cn}]，所以在匹配"Thai block中已经赋值的字符"时，[\p{InThai}&&\P{Cn}]也可以写作[\p{InThai}&&[^\p{Cn}]]。这样的改变并没有多少意义，只是它有助于说明一个通用的模式："Thai block中已赋值字符"比"所有Thai block中的字符，**减去未赋值的字符**"更好理解，于是我们知道[\p{InThai}&&[^\p{Cn}]]表示"\p{InThai}**减去**\p{Cn}"。

这样就回到了本节开头「[[a-z]&&[^aeiou]]」的例子，现在我们知道如何进行**字符组的减法**了。其模式为：「[*this* &&[^*that*]]」表示"[*this*]减去[*that*]"。我发现用&&和[^…]进行双重否定很难记忆，所以记住模式「[… &&[^…]]」就够了。

通过环视功能模拟字符组的集合运算 如果所使用的程序不支持字符组集合运算，但支持环视功能（☞133），则可以自己模拟集合运算。「[\p{InThai}&&[^\p{Cn}]]」可以用环视功能

写成「(?!\p{Cn})\p{InThai}」(注 13)。尽管它并不如内建的集合运算有效率，环视仍然是非常方便的做法。这个例子可以用 4 种不同的方式来实现（在.NET 中需要以 IsThai 替换 InThai☞125)。

```
(?!\p{Cn})\p{InThai}
(?=\P{Cn})\p{InThai}
\p{InThai}(?<!\p{Cn})
\p{InThai}(?<=\P{Cn})
```

POSIX“字符组”方括号表示法

我们通常所说的**字符组**，在 POSIX 标准中称为**方括号表达式**（bracket expression)。POSIX 中的术语“字符组”指的是在方括号表达式(注 14)内部使用的一种特殊的功能（special feature)，而我们可以认为它们是 Unicode 的字符属性的原型。

POSIX 字符组是 POSIX 方括号表达式使用的几种特殊元字符序列之一。比如[:lower:]表示当前 locale（☞87）中的所有小写字母。对英文文本来说，[:lower:]等于 a-z。因为整个序列只有在方括号表达式内才是有效的，所以对应的完整的字符组应该是「[[:lower:]]」。这种表示法的确很难看。但是，它比「[a-z]」更好用，因为它能包含 ö，ñ 之类当前 locale 中定义的“小写字母”。

POSIX 字符组的详细列表根据 locale 的变化而变化，但是下面这些通常都能支持：

- [:alnum:] 字母字符和数字字符。
- [:alpha:] 字母。
- [:blank:] 空格和制表符。
- [:cntrl:] 控制字符。
- [:digit:] 数字。
- [:graph:] 非空字符（即空白字符，控制字符之外的字符)。
- [:lower:] 小写字母。
- [:print:] 类似[:graph:]，但是包含空白字符。
- [:punct:] 标点符号。
- [:space:] 所有的空白字符（[:blank:]、换行符、回车符及其他)。
- [:upper:] 大写字母。
- [:xdigit:] 十六进制中容许出现的数字（例如 0-9a-fA-F)。

注 13：实际上，在 Perl 中，这个例子可能可以写作「\p{Thai}」，因为在 Perl 中\p{Thai}表是**字母表**，它包含未赋值的代码点。Thai 字母表和区块之前的其他差异很小。如果要确定某个字母表或者区块实际包含了哪些字符，最好的办法还是参考文档。在这里，字母表其实少了一些区块中包含的特殊字符。请参考 *http://unicode.org* 获得所有细节。

注 14：一般来说，本书中的“字符组（character class)”和“POSIX 括号表达式（POSIX bracket expression)”是指的同一种结构，而“POSIX 字符组”指的是下面介绍的特殊的类似范围表示法的字符组特性。

支持 Unicode 属性（☞121）的系统可能会在 Unicode 支持中加入这些 POSIX 结构。Unicode 属性结构更为强大，所以如果可能，这些结构应该有提供。

POSIX "collating 序列" 方括号表示法：`[[.span-ll.]]`

Local 可以包含对应的 *collating* **序列**，用来决定其中的字符如何排序。例如，在西班牙语中，按照惯例，ll 两个字母在排序时作为一个逻辑字符（例如在 *tortilla* 中），排在 l 和 m 之间，日尔曼语字母 ß 位于 s 和 t 中间，但是排序时类似它等价于两个字母 ss。这些规则可能用 collating 序列命名来表示，例如，span-ll 和 eszet。

collating 序列会把多个实体字符映射到单个逻辑字符，在 span-ll 的例子中，ll 被视为"一个字符"，来保持与 POSIX 正则引擎的兼容。也就是说，「[^abc]」能够匹配 'll' 两个字母。

collating 序列的元素可以包含在方括号表达式中，使用 [.….] 表示法：「torti[[.span-ll.]]a」匹配 tortilla。单个 collating 序列可以匹配组合而成的字符。此种情况下，方括号表达式可以匹配多个实体字符。

POSIX "字符等价类" 方括号表示法：`[[=n=]]`

有的 locale 定义了**字符等价类**，表示某些字符在进行排序之类的操作时应视为等价。例如，某 locale 可能定义了这样一个等价类 'n'，包含 n 和 ñ，或者是另一个等价类 'a'，包含 a、á 和 à。等价类的表示法类似 [:…:]，但是用等号取代冒号，我们可以在方括号表达式中引用这些等价类：「[[=n=][=a=]]」能够匹配刚才出现的任意一个字符。

如果一个字符等价类的名称只包含一个字母，但没有在 locale 中定义，则它默认就等于同样名字的 collating 序列。local 通常包含作为 collating 序列的普通字符 [.a.]、[.b.]、[.c.] 之类——如果没有定义特殊的等价类，「[[=n=][=a=]]」就等于「[na]」。

Emacs 语法类

GNU Emacs 不支持传统的「\w」、「\s」之类；相反，它使用特殊的序列来引用"语法类（syntax classes）"：

\s*char*　匹配 Emacs 语法类中 *char* 描述的字符。

\S*char*　匹配不在 Emacs 语法类中的字符。

「\sw」匹配"构成单词(word constituent)"的字符,而「\s-」匹配"空白字符"。在其他系统中,它们分别写作「\w」和「\s」。

Emacs 的特殊之处在于,在 Emacs 中,这些字符组包含的字符是可以临时更换的,所以,构成单词的字符组中的字符可以根据所编辑文本的变化而变化。

锚点及其他"零长度断言"

Anchors and Other "Zero-Width Assertions"

锚点和其他"零长度断言"并不会匹配实际的文本,而是寻找文本中的位置。

行/字符串的起始位置:^、\A

脱字符「^」匹配需要搜索的文本的起始位置,如果使用了增强的行锚点匹配模式(☞112),它还能匹配每个换行符之后的位置。在某些系统中,增强模式下「^」还能匹配 Unicode 的行终结符(☞109)。

如果可以使用,则无论在什么匹配模式下,「\A」总是能够匹配待搜索文本的起始位置。

行/字符串的结束位置:$、\Z 和\z

从下一页的表格 3-11 可以看出,"行结束位置(end of line)"的概念比行开头位置要复杂。在不同的工具软件中,「$」的意义也不同,不过最常见的意思是匹配目标字符串的末尾,也可以匹配**整个字符串末尾**的换行符之前的位置。后一种情况更为常见,它容许「s$」(匹配"以 s 结尾的行")来匹配'…s⏎',即以 s 和换行符结尾的行。

「$」的另两种常见的意思是,只匹配目标文本的结束位置,或是匹配任何一个换行符之前的位置。在某些 Unicode 系统中,这些规则中的换行符会被替换为 Unicode 的行终结符(☞109)(Java 为了处理 Unicode 的行终结符,为「$」设定了非常复杂的语意☞370)。

匹配模式(☞112)可以改变「$」的意义,匹配字符串中的任何换行符(或者是 Unicode 中的行终结符)。

如果支持,「\Z」通常表示"未指定任何模式下"「$」匹配的字符,通常是字符串的末尾位置,或者是在字符串末尾的换行符之前的位置。作为补充,「\z」只匹配字符串的末尾,而不考虑任何换行符。表 3-11 中列出了少数例外。

表 3-11：脚本语言中的行锚点

条目	Java	Perl	PHP	Python	Ruby	Tcl	.NET
正常情况							
^ 匹配字符串起始位置	✓	✓	✓	✓	✓	✓	✓
^ 匹配任意换行符					✓[2]		
$ 匹配字符串的结束位置	✓	✓	✓	✓	✓	✓	✓
$ 匹配字符串结尾的换行符	✓[1]	✓	✓	✓	✓		✓
$ 匹配**任何**换行符					✓[2]		
提供增强型行锚点模式（☞112）	✓	✓	✓	✓		✓	✓
在增强型行锚点模式中							
^ 匹配字符串的起始位置	✓	✓	✓	✓	N/A	✓	✓
^ 匹配任何换行符之后的位置	✓[1]	✓	✓	✓	N/A	✓	✓
$ 匹配字符串的末尾	✓	✓	✓	✓	N/A	✓	✓
$ 匹配**任何**换行符之前的位置	✓[1]	✓	✓	✓	N/A	✓	✓
\A 总是与普通的^一样	✓	✓	✓	✓	•[4]	✓	✓
\Z 总是与普通的$一样	✓[1]	✓	✓	•[3]	•[5]	✓	✓
\z 总是匹配字符串的末尾	✓	✓	✓	N/A	N/A	✓	✓

注：

在这些情况下，Sun 的 Java regex package 支持 Unicode 的**行终结符**（☞109）。

Ruby 的$和^能匹配字符串中的换行符，但是\A和\Z则不能。

Python 的\Z 只能匹配字符串的结束位置。

Ruby 的\A 与^不同，只能匹配字符串的起始位置。

Ruby 的\Z 与$不同，可以匹配字符串的结尾位置，或是字符串结尾的换行符之前的位置。

（请参考第 91 页的版本信息）

匹配的起始位置（或者是上一次匹配的结束位置）：\G

「\G」首先出现在 Perl 中。在使用/g（☞51）的匹配中，\G 对迭代操作非常有用，它能够匹配上一次匹配结束的位置。在第一次迭代时，「\G」匹配字符串的开头，与「\A」一样。

如果匹配不成功，「\G」的匹配会重新指向字符串的起始位置。这样，如果重复应用某个正则表达式，例如进行 Perl 的「s/…/…/g」操作，或者在其他语言中调用“找出所有匹配（match all)”函数，在匹配失败的同时，「\G」也会指向字符串的开头位置，这样以后进行其他类型的匹配操作便不受影响。

根据我的观察，Perl 的「\G」有 3 个值得注意而且很有用的方面：

- 「\G」的指向位置是**每个目标字符串**的属性，而不是设定这些位置的正则表达式的属性。也就是说，多个正则表达式可以依次对同一个字符串进行匹配，都使用上一轮匹配设定的「\G」。

- Perl 的正则运算符有一个选项（Perl 的/c 修饰符☞315），它规定了，如果匹配失败，不要重新设定「\G」，而只是保持之前的值不变化。如果结合上面那一点，就可以从某个位置开始尝试用多个正则表达式进行匹配，直到匹配能够成功，然后在下面的文本中继续寻找匹配。
- 「\G」对应的属性可以用与正则表达式无关的结构（Perl 的 pos 函数☞313）来检查和修改。可能有人希望设定这个位置来“规定”从什么位置开始寻找匹配，以及只从那个位置开始的匹配。同样，如果语言支持本条功能，而没有直接提供上一条功能，我们可以用本条功能来模拟。

下页的补充内容中有个例子展示了这些特性的用法。除了这些便捷之外，Perl 的「\G」还存在一个问题，即它必须出现在正则表达式的开头，这样才能正常工作。不过幸运的是，这似乎是最自然的用法。

之前匹配的结束位置，还是当前匹配的开始位置？

不同的实现方式之间存在一个区别，「\G」匹配的到底是“当前匹配的起始位置”还是“前一次匹配的结束位置”。在绝大多数情况下，这两者是等价的，所以大多数时候这个问题并不要紧。但也有些不常见的情况下，它们是有区别的。215 页有个例子说明了这种情况，不过最容易的还是用一个专门的例子来理解：把「x?」应用到‘abcde’。这个表达式能够在‘▲abcde’匹配成功，但其实它没有匹配任何文本。在进行全局查找-替换时，正则表达式会重复应用，每次处理上一次操作之后的文本，除非传动装置会做些特别的处理，“上次匹配完成的位置”总是它开始的位置。为了避免无穷循环，在这种情况下传动装置会强行前进到下一个字符（☞148），如果对‘abcde’应用 s/x?/!/g，结果就是‘!a!b!c!d!e!’。

传动装置这样处理会带来一个问题：“上一次匹配的终点”不等于“本次匹配的起点”。如果是这样，问题就来了：「\G」匹配哪个位置呢？在 Perl 中，对‘abcde’应用 s/**\G**x?/!/g 得到‘!abcde’，所以我们知道，在 Perl 中，「\G」只匹配上一次匹配的结束位置。如果传动装置自行驱动，Perl 的\G 肯定无法匹配。

另一方面，在其他某些工具软件中使用同样的查找-替换命令，会得到‘!a!b!c!d!e!’，也就是说\G 是在每次匹配的起始位置匹配成功，然后由传动装置进行驱动。

关于「\G」的匹配，也不能完全相信文档，微软的.NET 和 Sun 的 Java 文档，在我通知这两家公司之前，都是错误的（然后他们才修正）。现在的状态就是，PHP 和 Ruby 中的「\G」指向当前匹配的开头位置，而 Perl、java.util.regex 和.NET 匹配上一次匹配的结束位置。

Perl 中\G 的高级用法

下面的程序对$html 中的 HTML 代码进行简单的校验，确保其中只有少数几种 HTML 结构（例如<IMG>和<A>，以及>)。在 Yahoo!我用这种方法确保用户提交的 HTML 符合某些规范。

这段代码中最重要的就是 Perl 的 m/…/gc 匹配操作符，它会把这个正则表达式一次性应用到目标字符串，下一次匹配从上一次成功匹配之后的文本开始，如果匹配失败，也**不会**重新设定 position（☞315）。

这样，我们就能用包含多个表达式的“tag team”来检查字符串。从理论上说，它好像对所有这些表达式进行整体的迭代，但是这段程序的执行单位不是一次表达式而是一次匹配，而且能够临时新增或排除某些表达式。

```
my $need_close_anchor = 0; # 如果遇见了<A>而没有对应的</A>，则返回True

while (not $html =~ m/\G\z/gc) # 在整个字符串没有处理完之前
{
  if ($html =~ m/\G(\w+)/gc) {
    ...如果$1 中包含数字或单词——可以检查语言的规范性...
  } elsif ($html =~ m/\G[^<>&\w]+/gc) {
    # 其他非 HTML 代码——无关紧要
  } elsif ($html =~ m/\G<img\s+([^>]+)>/gci) {
    ...包含 image tag——可以检查它是否符合规范...

  } elsif (not $need_close_anchor and $html =~ m/\G<A\s+([^>]+)>/gci) {
    ...包含超链接，这里可以进行验证...

    $need_close_anchor = 1; # 我们现在需要的是</A>
  } elsif ($need_close_anchor and $html =~ m{\G</A>}gci){
    $need_close_anchor = 0; # 需求已经满足，不再容许出现
  } elsif ($html =~ m/\G&(#\d+<\w+);/gc){
    # 容许出现&gt;和&#123;之类的entity
  } else {# 此处完全无法匹配，必然有错误。记下当前位置，从HTML 中提取若干字符，报告错误
    my $location = pos($html); # 记下这段 HTML 的起始位置
    my ($badstuff) = $html =~ m/\G(.{1,12})/s;
    die "Unexpected HTML at position $location: $badstuff\n";
  }
}

# 确保没有孤立的 <A>
if ($need_close_anchor) {
  die "Missing final </A>"
}
```

单词分界符：\b、\B、\<、\> …

单词分界符的作用与行锚点一样，也是匹配字符串中的某些位置。单词分界符可以分为两类，一类中单词**起始位置**分界符和**结束位置**分界符是相同的（通常是\<和\>），另一类则以**统一**的分界符来匹配（通常是\b）。两类都提供了非单词分界符序列（通常是\B）。表 3-12 给出了一些例子。如果所使用的工具软件没有提供单独的起始位置和结束位置分界符，但支持环视功能，用户也可以用它来模拟那两种单词分界符。在下面的表格中，如果程序本身没有提供分开的单词分界符，我会列出实践中的做法。

单词分界符通常可以这样理解，这个位置的一边是“单词字符（word character)”，另一边则不是。每种工具软件对“单词字符”的理解都不一样，对单词边界的理解也是这样。如果单词分界符等于\w 当然好办，但很多时候事实并非如此。例如，在 PHP 和 java.util.regex 中，\w 只能匹配 ASCII 字符，而不是 Unicode 字符，所以在表格中我会使用带有 Unicode 单词属性\pL（这是「\p{L}」的缩略表示法☞121）的环视功能。

无论单词分界符怎么定义“单词字符”，单词分界符的测试通常只是简单的字符相邻测试。所有的正则引擎都不会对单词进行语意分析：它们认为“NE14AD8”是一个单词，而“M.I.T.”不是。

顺序环视(?=…)、(?!…)；逆序环视(?<=…)、(?<!…)

在前一章“使用环视功能在数值中插入逗号”（☞59）的例子中，我们已经介绍过顺序环视和逆序环视结构（统称为**环视**）。但关于它们还有很重要的一点没有介绍，那就是环视结构中能够出现什么样的表达式。大多数实现方式都限制了逆序环视中的表达式的长度（但是顺序环视则没有限制）。

Perl 和 Python 的限制是最严格的，逆序环视只能匹配固定长度的文本。使用 (?<!\w) 和 (?<!**this|that**) 不会出错 ，但是 (?<!**books?**) 和 (?<!^**\w+:**) 则不行，因为它们匹配的文本的长度是不确定的。某些情况下，(?<!**books?**) 可以重写为「(?<!book)(?<!books)」，尽管第一眼看上去它并不好理解。

表 3-12：若干工具软件中使用的单词分界符元字符

程序	单词开始…结束	单词分界符	非单词分界符
GNU awk	\<…\>	\y	\B
GNU *egrep*	\<…\>	\b	\B
GNU Emacs	\<…\>	\b	\B
Java	(?<!pL)(?=\pL)…(?<=pL)(?!\pL)	\b☒	\B☒
MySQL	[[:<:]]…[[:>:]]	[[:<:]]|[[:>]]	
.NET	(?<!\w)(?=\w)…(?<=\w)(?!\w)	\b	\B
Perl	(?<!\w)(?=\w)…(?<=\w)(?!\w)	\b	\B
PHP	(?<!pL)(?=\pL)…(?<=pL)(?!\pL)	\b☒	\B☒
Python	(?<!\w)(?=\w)…(?<=\w)(?!\w)	\b	\B
Ruby		\b☒	\B☒
GNU sed	\<…\>	\b	\B
Tcl	\m…\M	\y	\Y

☒表示只能对 ASCII 中的字符（或者是基于 locale 的 8 位编码数据）有效，即使该流派支持 Unicode 也是如此。

（请参考第 91 页的版本信息）

更高一层次的支持容许逆序环视中出现不同长度的多选分支，所以`(?<!books?)`可以写作`(?<!book|books)`。PCRE（因此也包括 PHP 中的 preg 套件）支持此功能。

最高层次的支持可以匹配任意长度的文本，只是其长度不能为无限。「`(?<!books?)`」可以直接使用，但是`(?<!^\w+:)`则不行，因为`\w+`能够匹配的长度没有限制。Sun 的 Java regex package 支持这样。

就问题本身来说，这三级支持其实是一样的，因为它们都表达同样的意思，尽管有的表达方式可能不太好看，而且对匹配的长度进行了严格的限制。中间一级只不过是"语法（syntactic sugar）"，表达方式更美观而已。而第四级支持容许逆序环视结构中的子表达式匹配任意长度的文本，甚至包括「`(?<!^\w+:)`」。微软的.NET 就支持这一级，它无疑是最棒的，但是如果运用不当，也可能带来严重的效率问题（如果逆序环视能够匹配任意长度的文本，引擎必须从字符串的起始位置开始检查逆序环视表达式，如果逆序环视是从长字符串的尾端开始的，这样就会浪费许多工夫）。

注释和模式修饰符

Comments and Mode Modifiers

在许多流派中，使用下面的结构，就能够在正则表达式内部，切换使用之前介绍的正则表达式模式和匹配模式（☞110）。

模式修饰符：(?*modifier*)，例如(?i)和(?-i)

现在，有许多流派容许在正则表达式中设定匹配模式（☞110）。常见的就是「(?i)」，它会启用不区分大小写的匹配，而「(?-i)」会停用此功能。例如，「<B>(?i)very(?-i)</B>」会对中间的「very」进行不区分大小写的匹配。而两端的 tag 仍然必须为大写。它可以匹配‘<B>VERY</B>’和‘<B>Very</B>’，但不能匹配‘Very’。

这个例子在大多数支持「(?i)」的系统中都可以运行，例如 Perl、PHP、java.util.regex、Ruby（注 15）和.NET。在 Python 和 Tcl 中则不行，因为它们不支持「(?-i)」。

除 Python 之外，大多数实现方式中，「(?i)」的作用范围都只限于括号内部（也就是说，在闭括号之后就失效）。所以，我们可以拿掉「(?-i)」，将整个不需要区分大小写的部分放在一个括号里，把「(?i)」放在最前面：「<B>**(?:**(?i)very**)**</B>」。

模式修饰符中能够出现的不只有‘i’。在大多数系统中，我们至少可以使用表 3-13 列出的修饰符。有的系统还提供了更多的选项。比如 PHP 就提供了少数其他选项（☞446），Tcl 也是如此（请参考文档）。

表 3-13：常见模式修饰符字母

字　母	模　式
i	不区分大小写的匹配模式（☞110）
x	宽松排列和注释模式（☞111）
s	点号通配模式（☞111）
m	增强的行锚点模式（☞112）

模式作用范围：(?*modifier*:...)，例如(?i:…)

如果所使用的系统支持模式修饰范围，这样前一节的例子可以更加简化。「(?i:…)」表示模式修饰符的作用范围只有在括号内有效。这样，「<B>(**?:**(**?i**)very)</B>」就可以化简为「<B>**(?i:**very**)**</B>」。

注 15：在 Ruby 中可以运行，但 Ruby 的(?i)有个 bug，即它有时不能正确处理用「|」分隔的小写多选分支（大写则没有问题）。

如果支持，这种格式一般可以应用于所有的模式修饰符字母。Tcl 和 Python 都支持「(?i)」格式，但是不支持「(?i:…)」格式。

注释：(?#…)和#…

某些流派支持用「(?#…)」添加注释。实际上，如果流派支持宽松排列和注释模式（☞111），就很少使用这种功能。不过，如果在字符串文字中很难插入换行符，用这种格式加入注释就非常方便，例如 VB.NET 就是如此（☞99，420）。

文字文本范围：\Q...\E

\Q…\E 是由 Perl 引入的，它会消除其中除\E 之外所有元字符的特殊含义（如果没有\E，就会一直作用到正则表达式末端）。其中的所有字符都会被当成普通文字文本来对待。如果在构建正则表达式时包含变量，此功能就非常有用。

举例来说，为了响应 Web 检索，我们可能希望把用户输入的内容保存在$query 中，然后使用 m/$query/i。但是，如果$query 包含某些字符，例如‘C:\WINDOWS\’，结果是运行时错误，因为这不是一个合法的正则表达式（最后有一个单独的反斜线）。

「\Q…\E」可以解决这个问题。如果在 Perl 中使用 m/\Q$query\E/i，则$query 就从‘C:\WINDOWS\’变成「C\:\\WINDOWS\\」，结果能找到用户期望的‘C:\WINDOWS\’。

但是在面向对象和程序式处理（☞95）中，这个特性的用处要打折扣。在构建需要用在正则表达式中的字符串时，有很方便的函数对这个值“上保险”，以便用在正则表达式中。例如，在 VB 中，我们可以使用 Regex.Escape(☞432)；PHP 提供了 preg_quote 函数(☞470)，Java 有 quote 方法（☞395）。

就我所知，支持「\Q…\E」的引擎只有 java.util.regex 和 PCRE(也包括 PHP 的 preg 套件)。请注意，我刚刚提到，这个功能是在 Perl 中引入的（而且我给出了 Perl 的例子），你可能觉得很奇怪，为什么刚刚没有提到 Perl。Perl 支持正则文字中的\Q…\E（也就是直接出现在程序中的正则表达式），但是不能在可能使用插值的内容和变量上使用。细节问题请参考第 7 章（☞290）。

在低于 1.6.0 的 Java 中，java.util.regex 对字符组中的「\Q…\E」支持是不可靠的，不建议使用。

分组，捕获，条件判断和控制

Grouping, Capturing, Conditionals, and Control

捕获/分组括号：(...)和\1，\2，…

普通的无特殊意义的括号通常有两种功能：分组和捕获。普通括号常见的形式是「(…)」，但有的流派中使用「\(…\)」，例如 GNU Emacs、*sed*、*vi* 和 *grep*。

如 41、43 和 57 页的图所示，捕获型括号的编号是按照开括号出现的次序，从左到右计算的。如果提供了反向引用，则这些括号内的子表达式匹配的文本可以在表达式的后面部分用「\1」、「\2」来引用。

括号的常用功能之一是从字符串中提取数据。括号中的子表达式匹配的文本（也可以称为“括号匹配文本（the text matched by the parentheses）”）在不同的程序中可以通过不同的方式来引用，例如 Perl 的`$1` 和`$2`（常见的错误是在正则表达式之外使用「\1」，这种形式只在 *sed* 和 *vi* 中能用）。下一页的表 3-14 说明了各种程序中，匹配完成之后访问文本的方法。它还说明了访问整个表达式匹配的文本，或者某一组捕获型括号所匹配文本的做法。

仅用于分组的括号：(?:...)

仅用于分组的括号「(?:…)」不能用来提取文本，而只能用来规定多选结构或者量词的作用对象。它们不会按照`$1`、`$2` 之类编号。在「(1|one)(?:and|or)(2|two)」匹配之后，`$1` 包含‘1’或者‘one’，`$2` 包含‘2’或者‘two’。只用于分组的括号也叫**非捕获型括号**(non-capturing parentheses)。

非捕获型括号的价值体现在好几个方面。它们能够把复杂的表达式变得清晰，这样读者不会担心在其他地方用到`$1` 会产生混乱。而且它们还有助于提高效率。如果正则引擎不需要记录捕获型括号匹配的内容，速度会更快，所用的内存也更少（第 6 章详细讲解效率问题）。

非捕获型括号的另一个用途是利用多个成分构建正则表达式。在第 76 页的例子中，`$HostnameRegex` 保存的是用来匹配主机名的正则表达式。如果使用它来提取主机名两端的空白，在 Perl 中是 `m/(\s*)$HostnameRegex(\s*)/`。然后`$1` 和`$2` 分别保存开头和结尾的空白，但结尾的空白其实是保存在`$4` 中的，因为`$HostnameRegex` 包含两组捕获型括号。

```
$HostnameRegex = qr/[-a-z0-9]+(\.[-a-z0-9]+)*\.(com|edu|info)/i;
```

表 3-14：若干工具软件及其中访问捕获文本的方法

程 序	完整的匹配	第一组括号匹配的文本
GNU *egrep*	N/A	N/A
GNU Emacs	(match_string 0) (replacement 字符串中为\&)	(match-string 1) (replacement 字符串中为\1)
GNU awk	Substr($*text*, RSTART, RLENGTH) (replacement 字符串中为\&)	\1 (在 gensub 替换中)
MySQL	N/A	N/A
Perl ☞41	$&	$1
PHP ☞450	$*matches*[0]	$*matches*[1]
Python ☞97	*MatchObj*.group(0)	*MatchObj*.group(1)
Ruby	$&	$1
GNU sed	& (只能在 replacement 字符串中使用)	\1(只能出现在 regex 和 replacement 中)
Java ☞95	*MatcherObj*.group()	*MatcherObj*.group(1)
Tcl	通过 regexp 命令设置为用户选择的变量	
VB.NET ☞96	*MatchObj*.Groups(0)	*MatchObj*.Groups(1)
C#	*MatchObj*.Gropus[0]	*MatchObj*.Groups[1]
vi	&	\1

（请参考第 91 页的版本信息）

如果这两组括号是非捕获型的，我们就可以按照直观的方式使用$HostnameRegex。另一种办法是使用命名捕获，尽管 Perl 没有提供这种功能，我们还是会介绍它。

命名捕获：(?<*Name*>...)

Python 和 PHP 的 preg 引擎，以及.NET 引擎，都能够为捕获内容命名。Python 和 PHP 使用的语法是「(?P<*name*>…)」，而.NET 使用「(?<*name*>…)」。我更喜欢.NET 的语法。下面是一个.NET 的例子：

```
「\b(?<Area>\d\d\d\)-(?<Exch>\d\d\d)-(?<Num>\d\d\d\d)\b」
```

在 Python/PHP 中是这样：

```
「\b(?P<Area>\d\d\d\)-(?P<Exch>\d\d\d)-(?P<Num>\d\d\d\d)\b」
```

这个表达式会用美国电话号码的各个部分"填充"*Area*、*Exch* 和 *Num* 命名的内容。然后我们可以通过名称来访问各个括号捕获的内容，例如在 VB.NET 和大多数.NET 语言中，可以使用 *RegexObj*.Groups("Area")，在 C#中使用 *RegexObj*.Groups["Area"]，在 Python 中使用 *RegexObj*.group("Area")，在 PHP 中使用$*matches*["Area"]。这样程序看起来更清晰。

如果要在正则表达式内部引用捕获的文本，.NET 中使用「\k<Area>」，Python 和 PHP 中使用「(?P=Area)」。

在 Pyhon 和.NET（但不包括 PHP）中，可以在同一个表达式中多次使用同样的命名。例如美国电话号码的区号部分的形式是‘(###)’或者‘###-’，为了匹配，我们可以使用（用.NET 语法）：「…(?:\((?<Area>\d\d\d)\)|(?<Area>\d\d\d)-)…」。无论哪一组匹配成功，都会把 3 位的区号保存到 *Area* 中。

固化分组：(?>…)

如果详细了解正则引擎的匹配原理（☞169），就很容易理解固化分组。在这里只说一点，就是一旦括号内的子表达式匹配之后，匹配的内容就固定下来（固化（atomic）下来无法改变），在接下来的匹配过程中不会变化，除非整个固化分组的括号都被弃用，在外部回溯中重新应用。下面这个简单的例子会帮助我们理解这种匹配的“固化”性质。

「¡.*!」能够匹配‘¡Hola!’，但是如果「.*」在固化分组「¡(?>.*)!」中就无法匹配。在这两种情况下，「.*」都会首先匹配尽可能多的内容（‘¡Hola!’），但是之后的「!」无法匹配，会强迫「.*」释放之前匹配的某些内容（最后的‘!’）。如果使用了固化分组，就无法实现，因为「.*」在固化分组中，它永远也不会“交还”已经匹配的任何内容。

尽管这个例子没有什么实际价值，固化分组还是有重要的用途。尤其是，它能够提高匹配的效率（☞171），而且能够对什么能匹配，什么不能匹配进行准确地控制（☞269）。

多选结构：…|…|…

多选结构能够在同一位置测试多个子表达式。每个子表达式称为一个**多选分支**（alternative）。符号「|」有很多称呼，不过“**或**（or）”和“**竖线**（bar）”最为常见。有的流派使用「\|」。

多选结构的优先级很低，所以「this and|or that」的匹配等价于「(this and)|(or that)」，而不是「this (and|or) that」，虽然 and|or 看上去是一个单位。

大多数流派都容许出现空的多选分支，例如「(this|that|)」。空表达式在任何情况下都能匹配，所以这个例子等于「(this|that)?」（注 16）。

注 16：认真考究起来，「(this|that|)」在逻辑上等价于「((?:this|that)?)」。它与语句「(this|that)?」的区别在于括号中是否包含“没有任何内容的匹配”，这个区别很细微，但是如果工具软件对匹配是否参与最终匹配会区别对待，就很重要。

POSIX 标准禁止出现空多选分支，*lex* 和大多数版本的 awk 也是如此。我认为，考虑到空多选分支的简便和清晰，保留它不无益处。Larry Wall 告诉我："我认为，保留它就好像在数学中保留 0 一样有意义"。

条件判断：(?if then |else)

这个结构容许用户在正则表达式中使用 if/then/else 判断。*if* 部分是特殊的条件表达式（a special kind of conditional expression），下文马上会有介绍。*then* 和 *else* 部分是普通的子表达式。如果 *if* 部分测试为真，则尝试 *then* 的表达式，否则尝试 *else* 部分（*else* 部分也可以不出现，果真如此的话，可以省略'|'）。

if 的种类因流派的不同而不同，但是大多数实现方式都容许在其中引用捕获的子表达式和环视结构。

测试对捕获型括号的特殊引用。如果 *if* 部分是一个括号中的编号，而对应编号的捕获型括号参与了匹配，其值为"true"。下面的例子匹配<IMG> tag，无论是是单独出现的，或者是在<A>…</A>中出现的。代码采用带注释的宽松排列格式，条件判断结构（这里的没有 *else* 部分）以粗体标注。

```
( <A\s+[^>]+> \s* )?    # 匹配开头的<A> tag，如果存在的话
<IMG\s+[^>]+>           # 匹配<IMG> tag
(?(1)\s*</A>)           # 匹配结尾的</A>，如果之前匹配过<A>
```

「(?(1)…)」测试中的(1)会测试第一组捕获型括号是否参与了匹配。"参与匹配"不等于"实际匹配了文本"，来看个简单的例子：

下面两种办法都可以匹配可能包含在"<…>"中的单词：「(<)?\w+(?(1)>)」可以，「(<?)\w+(?(1)>)」则不行。它们之间唯一的区别在于第一个问号出现的位置。在第一种（正确的）办法中，问号作用于整个捕获型括号，所以括号（以及包含的内容）不是必须匹配的。在第二个例子中，捕获型括号不是可选的——只有其中的「<」才是，所以无论'<'是否匹配了文本，它都"参与匹配"。也就是说，「(?(1)…)」中的 *if* 部分总是"true"。

如果能够使用命名捕获（☞138），就可以在括号中使用命名，而不是编号。

用环视做测试。完整的环视结构，例如「(?=…)」和「(?<=…)」，可以用于 *if* 测试。如果环视能够匹配，它返回"true"，执行 *then* 部分。否则会执行 *else* 部分。来看个专门设计的例子「(?(?<=NUM:)\d+|\w+)」，它会在「NUM:」之后的位置尝试匹配「\d+」，但是在其他位置尝试

使用「\w+」。环视条件判断以下画线标注。

条件判断的其他测试。Perl 提供了一种复杂的条件判断结构，容许在测试中使用任意 Perl 代码。返回值作为测试的值，根据它来判断 *then* 或者 *else* 部分是否应该尝试。详细信息请参考第 7 章的第 327 页。

匹配优先量词：*、+、?、{*num*，*num*}

量词（星号、加号、问号，以及区间元字符，它们能够限制作用对象的匹配次数）已经有过详细介绍。不过，需要注意的是，在某些工具中使用「\+」和「\?」来取代「+」和「?」。同样，在某些更老的工具中，量词不能限定反向引用，也不能限定括号。

区间：{*min*,*max*}或者\{*min*,*max*\}

区间可以被认为是“计数量词（counting quantifier）”，因为用户可以通过区间指定匹配成功所必须的下限和上限。如果只设置了一个数值（例如「[a-z]{3}」或者「[a-z]\{3\}」，依流派决定），匹配的次数就等于这个值。它等同于「[a-z][a-z][a-z]」（尽管两者的效率可能有所不同☞251）。

需要注意的是，不要认为「X{0,0}」的意思是“X 不能出现”。「X{0,0}」没有意义，因为它的意思是“不需要匹配「X」，也就是说实际上根本不需要进行任何尝试”。它基本等于「X{0,0}」不存在——如果存在 X，它也可以被正则表达式之后出现的某些元素匹配，所以这种做法是行不通的（注 17）。要实现“不容许存在”，请使用否定性环视。

忽略优先量词：*?、+?、??、{*num*,*num*}?

有的工具提供了不那么美观的量词：*?、+?、??和{*min*, *max*}?。这些是忽略优先的量词。量词在正常情况下都是“匹配优先（greedy）”的，匹配尽可能多的内容。相反，这些忽略优先的量词会匹配尽可能少的内容，只需要满足下限，匹配就能成功。其中的差异有深远的影响，详细的介绍在下一章（☞159）。

注 17：从理论上说，我关于{0,0}的论述是没错的。但是，实际的情况更糟糕，因为它会产生随机的结果！在许多程序（包括 GNU awk、GNU *grep* 以及老版本的 Perl）中，{0,0}就等价于*，而在其他许多程序（包括我曾见过的大多数版本的 *sed*，以及某些版本的 *grep*），中等同?。这真叫人抓狂！

占有优先量词：`*+`、`++`、`?+`、`{num, num}+`

这些量词目前只有 java.util.regex 和 PCRE（以及 PHP）提供，但是很可能会流行开来，占有**优先量词**类似普通的匹配优先量词，不过他们一旦匹配某些内容，就不会“交还”。它们类似固化分组，如果理解了基本的匹配过程，就很容易理解占有优先量词。

从某种意义上来说，占有优先的量词只是些表面工夫，因为它们可以用固化分组来模拟。「.++」与「(?>.+)」的结果完全一样，只是足够智能的实现方式能对占有优先量词进行更多的优化。

高级话题引导

Guide to the Advanced Chapters

我们已经熟悉了元字符、流派、语法包装（syntactic packaging）之类的概念，现在应该详细介绍本书开头提到的第三点了，也就是工具软件的正则引擎如何把一个正则表达式应用到文本当中。在第 4 章“**正则表达式的匹配原理**”中，我们会看到匹配引擎的实现方式如何影响匹配的**完成**、匹配的**内容**，以及匹配的**时间**。我们会详细考察这一切。学习完这些知识之后，你在调校复杂的正则表达式时会更有信心。第 5 章“**实用正则表达式技巧**”会用更复杂的例子巩固这些知识。

接下来是第 6 章“**打造高效率的正则表达式**”。了解了引擎的基本工作原理之后，你会学习到如何充分利用这些知识。第 6 章考察了正则表达式的陷阱——它们通常会导致意外的结果，然后教会读者真正运用书本上的知识。

第 4、5、6 三章是本书的核心。头三章只是为它们做铺垫，而且最后针对工具软件的章节以它们为基础。核心章节不容易阅读，但是我尽力避免使用数学、代数和其他我们不熟悉的概念。但是，就像任何高深的学问一样，潜心研究细节需要花费相当的工夫。

第4章 表达式的匹配原理

The Mechanics of Expression Processing

前一章在开头类比了正则表达式与汽车，余下的部分介绍了正则表达式的功能、特点以及其他相关信息。本章仍会使用这个类比来说明重要的正则引擎及其工作原理。

为什么需要了解这些原理呢？读者将会了解到，正则引擎分为很多种，最常用的引擎类型——Perl、Tcl、Python、.Net、Ruby、PHP，我见过的所有的Java正则包，以及其他语言使用的工作原理，基于此原理，构建正则表达式的**方式**决定了某个正则表达式**能否**匹配一个特定字符串，在**何处**匹配，以及匹配成功或报告失败的**速度**。如果你认为这些问题很重要，请阅读本章。

发动引擎

Start Your Engines!

现在我们来看看，引擎的类比能为我们提供多大帮助。引擎的价值在于，有了它，你不需要花多少气力就能从一个地方移动到另一个地方。驾驶员只需要放松或者听听音乐，发动机会完成余下的事情。它的主要任务就是驱动车轮，而驾驶员没必要关心它是如何工作的。是这样吗？



两类引擎
Two Kinds of Engines

设想一下驾驶电动汽车的情形？电动汽车已经诞生很久了，但它们不像汽油发动机驱动的汽车那样普及，因为电动汽车还不够成熟。如果你有辆电动汽车，请记住别给它加油。如果你的汽车采用汽油发动机，请务必远离烟火。电动机几乎总是“能够运行”，汽油机则需要多加保养。更换火花塞、空气过滤器，或者换用不同品牌的汽油，有可能大大提升发动机的效率。当然，也可能降低汽油机的性能，或者导致发动机罢工。不同引擎的工作原理也有不同，但目的都是驱动车轮。不过，如果你想开车去某个地方，还得把好方向盘，当然，这是题外话。

新的标准
New Standards

让我们看看添加一条新规范：加利福尼亚州的尾气排放标准（注 1）。一些发动机达到了加州的严格排放标准，一些则没有。这两类发动机并没有本质的不同，只是按标准划分为两类。这些标准规定了发动机尾气排放的成分，而并没有规定发动机应该怎样做才能达标。所以，现在我们可以把之前的两分法变为四分法：符合标准的电动机、不符合标准的电动机、符合标准的汽油机和不符合标准的汽油机。

回到原来的话题，我敢打赌，电动机不需要做多少改动就可以达标——标准只是“规定”尾气的成分，而电动机几乎没有尾气。相反，汽油机要达标可能就需要大的修改和更新。使用汽油发动机的驾驶员尤其需要注意汽油的型号——如果加错了油，他们就惹上大麻烦了。

标准的作用

更严格的排放标准是个好玩意儿，但驾驶员也需要考虑更多，同时更加小心（至少对汽油车来说如此）。不过坦白说，新标准对大多数人没有什么影响，因为其他州不会施行加州的标准。

所以你知道，四种类型的发动机其实可以分为三类：两类是汽油机，一类是电动机。虽然它们都是驱动车轮的，但你明白了其中的差异。你不知道的是，这堆复杂的玩意与正则表达式有什么关系！其实这里面的关系远比你能想象的要复杂。

注 1：加州对汽车尾气排放的限制非常严格。因此，美国市场上的不少汽车都标明“达到加州标准”或“未达到加州标准”。

正则引擎的分类

Regex Engine Types

正则引擎主要可以分为基本不同的两大类：一种是DFA（相当于之前说的电动机），另一种是NFA（相当于前面的汽油机）。我们很快就会知道DFA和NFA的具体含义，但是现在读者只需要知道这两个名字，就像"Bill"和"Ted"，"汽油机"和"电动机"一样。

DFA和NFA都有很长的历史，不过，正如汽油机一样，NFA的历史更长一些。使用NFA的工具包括.NET、PHP、Ruby、Perl、Python、GNU Emacs、*ed*、sec、*vi*、*grep*的多数版本，甚至还有某些版本的*egrep*和awk。而采用DFA的工具主要有*egrep*、awk、*lex*和*flex*。也有些系统采用了混合引擎，它们会根据任务的不同选择合适的引擎（甚至对同一表达式中的不同部分采用不同的引擎，以求得功能与速度之间的最佳平衡）。表4-1列出了少量常用的工具及其大多数版本使用的引擎。如果你最喜欢的工具没有名列其中，可以参考下一页的"测试引擎的类型"来找到答案。

表4-1：部分程序及其所使用的正则引擎

引擎类型	程 序
DFA	awk（大多数版本）、*egrep*（大多数版本）、*flex*、*lex*、MySQL、Procmail
传统型NFA	GNU Emacs、Java、*grep*（大多数版本）、*less*、*more*、.NET语言、PCRE library、Perl、PHP（所有三套正则库）、Python、Ruby、sed（大多数版本）、*vi*
POSIX NFA	*mawk*、Mortice Kern Systems' utilities、GNU Emacs （明确指定时使用）
DFA/NFA混合	GNU awk、GNU *grep*/*egrep*、Tcl

第3章已经讲过，NFA和DFA都发展了二十多年，产生了许多不必要的变体，结果，现在的情况比较复杂。POSIX标准的出台，就是为了规范这种现象，POSIX标准不但清楚地规定了前一章中提到的引擎应该支持的元字符和特性，还明确规定了**使用者期望由表达式获得的准确结果**。除开表面细节不谈，DFA（也就是电动机）显然已经符合新的标准，但是NFA风格的结果却与此不一，所以NFA需要修改才能符合标准。这样一来，正则引擎可以粗略地分为3类：

- DFA（符合或不符合POSIX标准的都属此类）。
- 传统型NFA。
- POSIX NFA。

这里提到的POSIX是匹配意义上的，也就是说，POSIX标准规定的某个正则表达式的应有行为（本章稍后部分将讨论）；而不是指POSIX标准引入的匹配特性。许多程序支持这些特性，但结果与POSIX规范不完全一致。

老式（功能极少的）程序，比如 *egrep*、*awk*、*lex* 之类，一般都是使用 DFA 引擎（电动机），所以，新的标准只是肯定了既有的情况，而没有大的改变。但是也存在一些汽油机版本的此类程序，如果它们需要达到 POSIX 标准，就需要做些修改。通过了加州排放标准测试（POSIX NFA）的汽油机能够产生符合标准的结果，但是这些必要的修改会增加保养的难度。以前，错位的火花塞也能应付着使用，但现在根本就点不着火。以前还能"凑合"的汽油，现在会弄得发动机砰砰乱响。不过，一旦掌握其中的门道，发动机就能平稳安静地运转了。

几句题外话

From the Department of Redundancy Department

现在，我请读者回过头去，重新思考关于引擎的故事。其中的每句话都涉及某些与正则表达式相关的事实。读第二遍会引起许多思考。尤其是，为什么说电动机（DFA 引擎）只是"能够运行"。什么影响了汽油机（NFA）？使用 NFA 引擎时，应该如何调整才能获得期望的结果？通过（排放）测试的 POSIX DFA 有什么特别之处？现实中"熄火的引擎"又是什么？

测试引擎的类型

Testing the Engine Type

工具所采用的引擎的类型，决定了引擎能够支持的特性以及这些特性的用途。所以，通常情况下，我们只需要几个测试用的表达式，就能判断出程序所使用的引擎类型（毕竟，如果你不能分辨引擎的类型，这种分类就没有意义）。现在，我并不期望读者理解下面的这些测试原理，我只是提供一些测试表达式，即使读者最喜欢使用的软件没有出现在表 4-1 之内，也可以判断出引擎的类型，继续阅读本章的其他内容。

是否传统型 NFA

传统型 NFA 是使用最广泛的引擎，而且它很容易识别。首先，看看忽略优先量词（☞141）是否得到支持？如果是，基本就能确定这是传统型 NFA。我们将要看到，忽略优先量词是 DFA 不支持的，在 POSIX NFA 中也没有意义。为了确认这一点，只需要简单地用正则表达式「nfa|nfa·not」来匹配字符串 'nfa·not'，如果只有 'nfa' 匹配了，这就是传统型 NFA。如果整个 'nfa not' 都能匹配，则此引擎要么是 POSIX NFA，要么是 DFA。

DFA 还是 POSIX NFA

某些情况下，DFA 与 POSIX NFA 的区别是很明显的。DFA 不支持捕获型括号（capturing parentheses）和回溯（backreferences），这一点有助于判断，不过，也存在同时使用两种引擎的混合系统，在这种系统中，如果没有使用捕获型括号，就会使用 DFA。

下面这个简单的测试能说明很多问题，用「X(.+)+X」来匹配形如‘=XX======================’的字符串，例如使用 *egrep* 命令：

```
echo =XX========================================= | egrep 'X(.+)+X'
```

如果执行需要花很长时间，就是 NFA（如果上一项测试显示这不是传统型 NFA，那么它肯定是 POSIX NFA）。如果时间很短，就是 DFA，或者是支持某些高级优化的 NFA。如果显示堆栈超溢（stack overflow），或者超时退出，那么它是 NFA 引擎。

匹配的基础

Match Basics

在了解不同引擎的差异之前，我们先看看它们的相似之处。汽车的各种动力系统在某些方面是一样的（或者说，从实用的角度考虑，它们是一样的），所以，下面的范例也能够适用于所有的引擎。

关于范例

About the Examples

本章关注的是一般的提供所有功能的正则引擎，所以，某些程序并不能完全支持它们。在本书所说的汽车里，机油油尺（dipstick）可能挨在机油滤清器（oil filter）的左边，而在读者那里，它却装在分电盘盖（distributor cap）的后面。不过，读者要做的只是理解这些概念，能够使用和维护自己最喜欢（以及他们最感兴趣）的正则表达式包。

在大部分例子中，我仍然使用 Perl 表示法，虽然我偶尔会用一些其他的表示法来提醒读者，表示法并不重要，我们讨论的问题与程序和表示法不属于一个层次。为节省篇幅，如果读者遇到不熟悉的构建方式，请查阅第 3 章（☞114）。

本章详细阐释了匹配执行的实际流程。理想的情况是，所有的知识都能归纳为几条容易记忆的简单规则，使用者不需要了解这些规则包含的原理。很不幸，事实并非如此。整个第 4 章只能列出两条普适的原则：

1. 优先选择最左端（最靠开头）的匹配结果。

2. 标准的匹配量词（「*」、「+」、「?」和「{m,n}」）是匹配优先的。

在本章中，我们将考察这些规则，它们的结果，以及其他许多内容。首先我们详细讨论第一条规则。

规则 1：优先选择最左端的匹配结果

Rule 1: The Match That Begins Earliest Wins

根据这条规则，起始位置最靠左的匹配结果总是优先于其他可能的匹配结果。这条规则并没有规定优先的匹配结果的长度（稍后将会讨论），而只是规定，在所有可能的匹配结果中，优先选择开始位置最左端的。实际上，因为可能有多个匹配结果的起始位置都在最左端，也许我们应该把这条规则中的“**某个**匹配结果（a match）”改为“**该**匹配结果（the match）”，不过这听起来有些别扭。

这条规则的由来是：匹配先从需要查找的字符串的起始位置尝试匹配。在这里，“尝试匹配（attempt）”的意思是，在当前位置测试整个正则表达式（可能很复杂）能匹配的每样文本。如果在当前位置测试了所有的可能之后不能找到匹配结果，就需要从字符串的第二个字符之前的位置开始重新尝试。在找到匹配结果以前必须在所有的位置重复此过程。只有在尝试过所有的起始位置（直到字符串的最后一个字符）都不能找到匹配结果的情况下，才会报告“匹配失败”。

所以，如果要用「ORA」来匹配 FLORAL，从字符串左边开始第一轮尝试会失败（因为「ORA」不能匹配 FLO），第二轮尝试也会失败（「ORA」同样不能匹配 LOR），从第三个字符开始的尝试能够成功，所以引擎会停下来，报告匹配结果 FLORAL。

如果不了解这条规则，有时候就不能理解匹配的结果。例如，用「cat」来匹配：

```
The dragging belly indicates that your cat is too fat.
```

结果是 indicates，而不是后来出现的 cat。单词 cat 是能够被匹配的，但 indi**cat**es 中的 cat 出现的更早，所以得到匹配的是它。对于 *egrep* 之类的程序来说，这种差别是无关紧要的，因为它只关心“**是否**”能够匹配，而不是“**在哪里**”匹配。但如果是进行其他的应用，例如查找和替换，这种差别就很重要了。

这里有一个小测验（应该不困难）：如果用「fat|cat|belly|your」来匹配字符串‘The dragging belly indicates that your cat is too fat.’，结果是什么呢？❖请看下一页。

“传动装置（transmission）”和驱动过程（bump-along）

或许汽车变速箱（译注 1）的例子有助于理解这条规则，驾驶员在换档时，变速箱负责连接引擎和动力系统。引擎是真正产生动力的地方（它驱动曲轴），而变速箱把动力传送到车轮。

译注 1：此处的“变速箱”就是上文中的“传动装置”，为了保持译文通畅，在涉及汽车时译为“变速箱”，在涉及正则表达式时译为“传动装置”。

传动装置的主要功能：驱动

如果引擎不能在字符串开始的位置找到匹配的结果，传动装置就会推动引擎，从字符串的下一个位置开始尝试，然后是下一个，再下一个，如此继续。不过，如果某个正则表达式是以“字符串起始位置锚点（start-of-string anchor）”开头的，传动装置就会知道，不需要更多的尝试，因为如果能够匹配，结果肯定是从字符串的头部开始的。在第 6 章中，我们会讲解这一点，以及更多的内部优化措施。

引擎的构造

Engine Pieces and Parts

所有的引擎都是由不同的零部件组合而成的。如果对这些零件缺乏了解，也就不可能真正理解引擎的工作原理。正则引擎中的这些零件分为几类——文字字符（literal characters）、量词（qualifiers）、字符组（character classes）、括号，等等，我们在第 3 章介绍过（☞114）。这些零件的组合方式（以及正则引擎对它们的处理方式）决定了引擎的特性，所以，这些零件的组合方式，以及它们之间的配合，是我们主要关注的东西。首先，让我们来看看这些零件：

文字文本（Literal Text）例如 a、*、!、枝…

对于非元字符的文字字符，尝试匹配时需要考虑就是“这个字符与当前尝试的字符相同吗？”。如果一个正则表达式只包含纯文本字符，例如「usa」，那么正则引擎会将其视为：一个「u」，接着一个「s」，接着一个「a」。进行不区分大小写的匹配时的情况要复杂一点，因为「b」能够匹配 B，而「B」也能匹配 b，不过这仍然不难理解（Unicode 的情况稍微复杂一些☞110）。

字符组、点号、Unicode 属性及其他

通常情况下，字符组、点号、Unicode 属性及其他的匹配是比较简单的：无论字符组的长度是多少，它都只能匹配一个字符（注 2）。

点号可以很方便地表示复杂的字符组，它几乎能匹配所有字符，所以它的作用也很简单，其他的简便方式还包括「\w」,「\W」和「\d」。

捕获型括号

用于捕获文本的括号（而不是用于分组的括号）不会影响匹配的过程。

注 2：其实，正如我们在前一章看到的（☞128），POSIX 的 collating 序列能够匹配多个字符，但这并不常见。同样，在进行不区分大小写的匹配时，某些 Unicode 字符可以匹配多个字符（☞110），尽管大多数实现并不支持此功能。

测验答案

◆ 148 页测验的答案

请记住，正则表达式的每一次尝试都要进行到底，所以「fat|cat|belly|your」用来匹配‘The dragging belly indicates your cat is too fat’的结果不是 fat，尽管「fat」在所有可能选项中列在最前头。

当然，正则表达式应该也能够匹配 fat 和其他可能，但它们都不是最先出现的匹配结果（除现在最左边的结果），所以不会被选择。在进行下一轮尝试之前，正则表达式的所有可能都会尝试，也就是说，在移动之前，「fat」、「cat」、「belly」和「your」都必须尝试。

锚点（e.g.，「^」「\z」「(?<=\d)」…）

锚点可以分为两大类：简单锚点（^、$、\G、\b、…☞129）和复杂锚点（例如顺序环视和逆序环视☞133）。简单锚点之所以得名，就在于它们只是检查目标字符串中的特定位置的情况（^、\Z…），或者是比较两个相邻的字符（\<、\b、…）。相反，复杂锚点（环视）能包含任意复杂的子表达式，所以它们也可以任意复杂。

非“电动”的括号、反向引用和忽略优先量词

虽然本章希望讲解的是引擎之间的相似之处，但为了方便读者理解本章余下的内容，这里必须指出一些有意义的差异。捕获括号（以及相应的反向引用和$1 表示法）就像汽油添加剂一样——它们只对汽油机（NFA）起作用，对电动机（DFA）不起作用。忽略优先量词也是如此。这种情况是由 DFA 的工作原理决定的（注 3）。这解释了，为什么 DFA 引擎不支持这些特性。读者会看到，awk、*lex* 和 *egrep* 都不支持反向引用和$1 功能（表示法）。

也许读者会注意到，GNU 版本的 *egrep* 确实支持反向引用。这是因为它包含了两台不同的引擎。它首先使用 DFA 查找可能的匹配结果，再用 NFA（支持包括反向引用在内的所有特性）来确认这些结果。接下来，我们将看到 DFA 不能支持反向引用及捕获括号的原因，以及这种引擎能够存在的理由（DFA 有很多显著的优势，例如匹配速度非常快）。

注 3：这并非是说，我们不能糅合两种引擎的长处以求最佳。请参考第 182 页的补充内容。

规则 2：标准量词是匹配优先的

Rule 2: The Standard Quantifiers Are Greedy

至今为止，我们看到的特性都非常易懂。但仅仅靠它们还很不够——要完成复杂点的任务，就需要使用星号、加号、多选结构之类功能更强大的元字符。要彻底理解这些功能，需要学习更多的知识。

读者首先需要记住的是，标准匹配量词（?、*、+，以及{*min, max*}）都是“**匹配优先**（greedy）”的。如果用这些量词来约束某个表达式，例如「(expr)*」中的「(expr)」、「a?」中的「a」和「[0-9]+」中的「[0-9]」，在匹配成功之前，进行尝试的次数是存在上限和下限的。在前面的章节中我们已经提到过这一点——而规则 2 表明，这些尝试总是希望获得最长的匹配（一些工具提供了其他的匹配量词，但是本节只讨论标准的匹配优先量词）。

简而言之，标准匹配量词的结果“可能”并非所有可能中最长的，但它们总是尝试匹配尽可能多的字符，直到**匹配上限**为止。如果最终结果并非该表达式的所有可能中最长的，原因肯定是匹配字符过多导致匹配失败。举个简单的例子：用「\b\w+s\b」来匹配包含‘s’的字符串，比如说‘regexes’，「\w+」完全能够匹配整个单词，但如果用「\w+」来匹配整个单词，「s」就无法匹配了。为了完成匹配，「\w+」必须匹配‘regexes’，把最后的「s\b」留出来。

如果表达式的其他部分能够成功匹配的唯一条件是，匹配优先的结构不匹配任何字符，在容许零匹配（译注 2）的情况下（例如使用星号、问号，或者{0, *max*}，这是没有问题的。不过，这种情况只有在表达式的后续部分强迫下才能发生。匹配优先量词之所以得名，是因为它们总是（或者，至少是尝试）匹配多于匹配成功下限的字符。

匹配优先的性质可以非常有用（有时候也非常讨厌）。它可以用来解释「[0-9]+」为什么能匹配 March·1998 中的所有数字。1 匹配之后，实际上已经满足了成功的下限，但此正则表达式是匹配优先的，所以它不会停在此处，而会继续下去，继续匹配‘998’，直到这个字符串的末尾（因为「[0-9]」不能匹配字符串最后的空档，所以会停下来）。

邮件主题

显然，这种匹配方式并非只能用于匹配数字。举例来说，如果我们需要判断 E-mail 的 header 中的某行字符是否标题行（subject line）。前面（☞55）我们已经说过，可以用「^Subject:」来实现。不过，如果使用「^Subject:·(.*)」，我们就能在之后的程序中使用捕获型括号来访问主题的内容（例如 Perl 中的$1）（译注 3）。

译注 2：zero match，即不匹配任何字符也能成功的匹配。

译注 3：这个例子用捕获型括号来讲解匹配优先，所以它只适用于 NFA（只有 NFA 支持捕获型括号）。不过，匹配优先的特性对所有引擎都是一样的，包括不支持捕获的 DFA。

在探讨「.*」匹配邮件主题之前，请读者记住，一旦「^Subject: ·」能够部分匹配，整个正则表达式就一定能够全部匹配。因为「^Subject: ·」之后没有字符会导致表达式匹配失败：「.*」永远不会失败，因为“不匹配任何字符”也是「.*」的可能结果之一。

那么，为什么要添加「.*」呢？这是因为我们知道，星号是匹配优先的，它会用点号匹配尽可能多的字符，所以我们用它来“填充” $1。事实上，括号并没有影响正则表达式的匹配过程，在本例中，我们只是用它们来包括「.*」匹配的字符。

「.*」到达字符串的末尾之后点号不能继续匹配，所以星号最终停下来，尝试匹配表达式中的下一个元素（尽管「.*」无法继续匹配了，但下面的子表达式或许能够继续匹配）。不过，因为本例中不存在后面的元素，到达表达式的末尾之后，我们就获得了成功的匹配结果。

过度的匹配优先

现在让我们回过头去看“尽可能匹配”的匹配优先量词。如果我们在上面的例子中增加一个「.*」，把正则表达式写作「^Subject:·(.*).*」，结果会是如何呢？答案是，没有变化。开头的「.*」（括号中的）会霸占整个标题的文本，而不给第二个「.*」留下任何字符。而第二个「.*」的匹配失败并不要紧，因为「.*」不匹配任何字符也能成功。如果我们给第二个「.*」也加上括号，$2 将会是空白。

这是否说明，在正则表达式中，「.*」的部分没有机会匹配任何字符呢？答案显然是否定的。就像我们在「\w+s」这个例子中看到的，如果进行全部匹配必须这样做，表达式中的某些部分可能“强迫”之前匹配优先的部分“释放”（或者说“交还（unmatch）”）某些字符。

「^.*([0-9][0-9])」或许是个有用的正则表达式，它能够匹配一行字符的最后两位数字，如果有的话，然后将它们存储在$1 中。下面是匹配的过程：「.*」首先匹配整行，而「[0-9][0-9]」是必须匹配的，在尝试匹配行末的时候会失败，这样它会通知「.*」：“嗨，你占的太多了，交出一些字符来吧，这样我没准能匹配。”匹配优先组件首先会匹配尽可能多的字符，但为了整个表达式的匹配，它们通常需要“释放”一些字符（抑制自己的天性）。当然，它们并不“愿意”这样做，只是不得已而为之。当然，“交还”绝不能破坏匹配成立必须的条件，比如加号的第一次匹配。

明白了这一点，我们来看「^.*([0-9][0-9])」匹配‘about·24·characters·long’的过程。「.*」匹配整个字符串以后，第一个「[0-9]」的匹配要求「.*」释放一个字符‘g’(最后的字符)。但是这并不能让「[0-9]」匹配，所以「.*」必须继续“交还”字符，接下来交还的字符是‘n’。如此循环 15 次，直到「.*」最终释放‘4’为止。

不幸的是，即使第一个「[0-9]」能够匹配‘4’，第二个「[0-9]」仍然不能匹配。为了匹配整个正则表达式，「.*」必须再次释放一个字符，这次是‘2’，由第一个「[0-9]」匹配。现在，‘4’能够由第二个「[0-9]」匹配，所以整个表达式匹配的是‘about·24·char…’，$1 的值是‘24’。

先来先服务

如果用「^.*[0-9]+」来匹配一行的最后两个数字，期望匹配的不止是最后两位数字，而是最后的整个数，结果会是多长呢？如果用它来匹配‘Copyright 2003.’，结果是什么？◆答案在下一页。

深入细节

在这里必须澄清一些东西。因为“「.*」必须交还…”或者“「[0-9]」迫使…”之类的说法或许容易引起混淆。我使用这些说法是因为它们易于理解，而且跟实际的结果一致。不过，事情的真相是由基本的引擎类型决定——是 DFA，还是 NFA。现在我们就来看这些。

表达式主导与文本主导

Regex-Directed Versus Text-Directed

DFA 和 NFA 反映了将正则表达式在应用算法上的根本差异。我把对应汽油机的 NFA 称为“表达式主导（regex-directed）”引擎，而对应电动机的 DFA 称为“文本主导（text-directed）”引擎。

NFA 引擎：表达式主导

NFA Engine: Regex-Directed

我们来看用「to(nite|knight|night)」匹配文本‘…tonight…’的一种办法。正则表达式从「t」开始，每次检查一部分（由引擎查看表达式的一部分），同时检查“当前文本（current text）”是否匹配表达式的当前部分。如果是，则继续表达式的下一部分，如此继续，直到表达式的所有部分都能匹配，即整个表达式能够匹配成功。

测验答案

◆ 153 页测验的答案

用「^.*([0-9]+)」匹配‘copyright 2003.’，括号会捕获到什么？

这个表达式的本意是捕获整个数字 2003，但结果并非如此。之前已经说过，为了满足「[0-9]+」的匹配，「.*」必须交还一些字符。在这个例子中，释放的字符是最后的‘3’和点号，之后‘3’能够由「[0-9]」匹配。「[0-9]」由「+」量词修饰，所以现在还只做到了最小的匹配可能，现在它遇到了‘.’，找不到其他可以匹配的字符。

与之前不同，此时没有“**必须**”匹配的元素，所以「.*」不会被迫交出 0。否则，「[0-9]+」应当心存感激，接受匹配优先元素的馈赠，但请记住“先来先服务”原则。匹配优先的结构只会在被迫的情况下交还字符。所以，最终$1 的值是‘3’。

如果读者觉得难以理解，不妨这样想，「[0-9]+」和「[0-9]*」差不多，而本例中「[0-9]*」和「.*」是一样的。用它来替换原来的表达式「^.*([0-9]+)」，我们得到「^.*(.*)」，这与 152 页的「^Subject:·(.*).*」很相似，第二个「.*」不会匹配任何字符。

在「to(nite|knight|night)」的例子中，第一个元素是「t」，它将会重复尝试，直到在目标字符串中找到‘t’为止。之后，就检查紧随其后的字符是否能由「o」匹配，如果能，就检查下面的元素。在本例中，“下面的元素”指「(nite|knight|night)」它的真正含义是“「nite」或者「knight」或者「night」”。引擎会依次尝试这 3 种可能。我们（具有高级神经网络的人类）能够发现，如果待匹配的字符串是 tonight，第三个选择能够匹配。不论神经学起源（☞85）如何，表达式主导的引擎必须完全测试，才能得出结论。

尝试「nite」的过程与之前一样：“尝试匹配「n」，然后是「i」，然后是「t」，最后是「e」。”如果这种尝试失败——就像本例，引擎会尝试另一种可能，如此继续下去，直到匹配成功或是报告失败。表达式中的控制权在不同的元素之间转换，所以我称它为“表达式主导”。



NFA 引擎在操作上的优点

实质上，在表达式主导的匹配过程中，每一个子表达式都是独立的。这不同于反向引用，子表达式之间不存在内在联系，而只是整个正则表达式的各个部分。在子表达式与正则表达式的控制结构（多选分支、括号以及匹配量词）的层级关系（layout）控制了整个匹配过程。

因为 NFA 引擎是正则表达式主导的，驾驶员（也就是编写表达式的人）有充足的机会来实现他/她期望的结果（第 5 章和第 6 章将会告诉读者，如何正确高效地实现目标）。现在看起来，这点还有些模糊，但过一段就会变清晰。

DFA 引擎：文本主导

DFA Engine: Text-Directed

与表达式主导的 NFA 不同，DFA 引擎在扫描字符串时，会记录“当前有效（currently in the works)”的所有匹配可能。具体到 `tonight` 的例子，引擎移动到 `t` 时，它会在当前处理的匹配可能中添加一个潜在的可能：

字符串中的位置	正则表达式中的位置
`after…t▴onight…`	可能的匹配位置：「`t▴o(nite|knight|night)`」

接下来扫描的每个字符，都会更新当前的可能匹配序列。继续扫描两个字符以后的情况是：

字符串中的位置	正则表达式中的位置
`after…toni▴ght…`	可能的匹配位置：「`to(ni▴te|knight|ni▴ght)`」

有效的可能匹配变为两个（`knight` 被淘汰出局）。扫描到 `g` 时，就只剩下一个可能匹配了。当 `h` 和 `t` 匹配完成后，引擎发现匹配已经完成，报告成功。

我称这种方式为“文本主导”，是因为它扫描的字符串中的每个字符都对引擎进行了控制。在本例中，某个未完成的匹配也许是任意多个（只要可行）匹配的开始。不合适的匹配可能在扫描后继文字时会被去除。在某些情况下，“处理中的未终结匹配（partial match in progress)”可能就是一个完整的匹配。例如正则表达式「`to(…)?`」，括号内的部分并不是必须出现的，但考虑到匹配优先的性质，引擎仍然会尝试匹配括号内的部分。匹配过程中，在尝试括号内的部分时，完整匹配（‘`to`’）已经保留下来，以应付括号中的内容无法匹配的情况。

如果引擎发现，文本中出现的某个字符会令所有处理中的匹配可能失效，就会返回某个之前保留的完整匹配。如果不存在这样的完整匹配，则要报告在当前位置无法匹配。

第一想法：比较NFA与DFA

First Thoughts: NFA and DFA in Comparison

如果读者根据上面介绍的知识比较NFA和DFA，可能会得出结论：一般情况下，文本主导的DFA引擎要快一些。正则表达式主导的NFA引擎，因为需要对同样的文本尝试不同的子表达式匹配，可能会浪费时间（就好像上面例子中的3个分支）。

这个结论是对的。在NFA的匹配过程中，目标文本中的某个字符可能会被正则表达式中的不同部分重复检测（甚至有可能被同一部分反复检测）。即使某个字表达式能够匹配，为了检查表达式中剩下的部分，找到匹配，它也可能需要再一次应用（甚至可能反复多次）。单独的子表达式可能匹配成功，也可能失败，但是，直到抵达正则表达式的末尾之前，我们都无法确知全局匹配成功与否（也就是说"不到最后关头不能分胜负(It's not over until the fat lady sings)"，但这句话又不符合本段的语境）。相反，DFA引擎则是**确定型的**（deterministic）——目标文本中的每个字符只会检查（最多）一遍。对于一个已经匹配的字符，你无法知道它是否属于最终匹配（它可能属于最终会失败的匹配），但因为引擎同时记录了所有可能的匹配，这个字符只需要检测一次，如此而已。

正则表达式引擎所使用的两种基本技术，都对应有正式的名字：**非确定型有穷自动机**（NFA）和**确定型有穷自动机**（DFA）。这两个名字实在是太饶舌，所以我坚持只用DFA和NFA。下文中不会出现它们的全称了（注4）。

用户需要面对的结果

因为NFA具有表达式主导的特性，引擎的匹配原理就非常重要。我已经说过，通过改变表达式的编写方式，用户可以对表达式进行多方面的控制。拿`tonight`的例子来说，如果改变表达式的编写方式，可能会节省很多工夫，比如下面这3种方式：

- 「`to(ni(ght|te)|knight)`」
- 「`tonite|toknight|tonight`」
- 「`to(k?night|nite)`」

注4：我倒是希望能讲解这两个名字背后的理论，可惜我不知道该如何做。我已经暗示，"**确定型**"这个名词是很重要的，但是，我们只需要懂得实际的效果，而这套理论的大部分内容与本书无关。读完本章，你会发现事实就是如此。

给出任意文本，这 3 个表达式都可以捕获相同的结果，但是它们以不同的方式控制引擎。现在，我们还无法分辨这 3 者的优劣，不过接下来会看到。

DFA 的情况相反——引擎会同时记录所有的匹配选择，因为这 3 个表达式最终能够捕获的文本相同，在写法上的差异并无意义。取得一个结果可能有上百种途径，但因为 DFA 能够同时记录它们（有点神奇，待稍后详述），选择哪一个表达式并无区别。对纯粹的 DFA 来说，即使「abc」和「[aa-a](b|b{1}|b)c」看来相差巨大，但其实是一样的。

如果要描述 DFA，我能想到的特征有：

- DFA 匹配很迅速。
- DFA 匹配很一致。
- 谈论 DFA 匹配很恼人。

最终我会展开这 3 点。

因为 NFA 是表达式主导的，谈论它是件很有意思的事情。NFA 为创造性思维提供了丰富的施展空间。调校好一个表达式能带来许多收益，调校不好则会带来严重后果。这就好比发动机的熄火和点不着火，他们并不只是汽油发动机的专利。为了彻底弄明白这个问题，我们来看 NFA 最重要的部分：**回溯**（backtracking）。

回溯

Backtracking

NFA 引擎最重要的性质是，它会依次处理各个子表达式或组成元素，遇到需要在两个可能成功的可能中进行选择的时候，它会选择其一，同时记住另一个，以备稍后可能的需要。

需要做出选择的情形包括量词（决定是否尝试另一次匹配）和多选结构（决定选择哪个多选分支，留下哪个稍后尝试）。

不论选择那一种途径，如果它能匹配成功，而且正则表达式的余下部分也成功了，匹配即告完成。如果正则表达式中余下的部分最终匹配失败，引擎会知道需要**回溯**到之前做出选择的地方，选择其他的备用分支继续尝试。这样，引擎最终会尝试表达式的所有可能途径（或者是匹配完成之前需要的所有途径）。

真实世界中的例子：面包屑

A Really Crummy Analogy

回溯就像是在道路的每个分岔口留下一小堆面包屑。如果走了死路，就可以照原路返回，直到遇见面包屑标示的尚未尝试过的道路。如果那条路也走不通，你可以继续返回，找到下一堆面包屑，如此重复，直到找到出路，或者走完所有没有尝试过的路。

在许多情况下，正则引擎必须在两个（或更多）选项中做出选择——我们之前看到的分支的情况就是一例。另一个例子是，在遇到「…x?…」时，引擎必须决定是否尝试匹配「x」。对于「…x+…」的情况，毫无疑问，「x」至少尝试匹配一次——因为加号要求必须匹配至少一次。第一个「x」匹配之后，**此要求已经满足**，需要决定是否尝试下一个「x」。如果决定进行，还要决定是否匹配第三个「x」，第四个「x」，如此继续。每次选择，其实就是洒下一堆“面包屑”，用于提示此处还有另一个可能的选择（目前还不能确定它能否匹配），保留起来以备用。

一个简单的例子

现在来看个完整的例子，用先前的「to(nite|knight|night)」匹配字符串‘hot·tonic·tonight!’（看起来有点无聊，但是个好例子）。第一个元素「t」从字符串的最左端开始尝试，因为无法匹配‘h’，所以在这个位置匹配失败。传动装置于是驱动引擎向后移动，从第二个位置开始匹配（同样也会失败），然后是第三个。这时候「t」能够匹配，接下来的「o」无法匹配，因为字符串中对应位置是一个空格。至此，本轮尝试宣告失败。

继续下去，从…▲tonic…开始的尝试则很有意思。to匹配成功之后，剩下的3个多选分支都成为可能。引擎选取其中之一进行尝试，留下其他的备用（也就是洒下一些面包屑）。在讨论中，我们假定引擎首先选择的是「nite」。这个表达式被分解为“「n」+「i」+「t」+「e」”，在…toni▲c…遭遇失败。但此时的情况与之前不同，这种失败并不意味着整个表达式匹配失败——因为仍然存在没有尝试过的多选分支（就好像是，我们仍然可以找到先前留下的面包屑）。假设引擎然后选择「knight」，那么马上就会遭遇失败，因为「k」不能匹配‘n’。现在只剩下最后的选项「night」，但它不能失败。因为「night」是最后尝试的选项，它的失败也就意味着整个表达式在…▲tonic…的位置匹配失败，所以传动机构会驱动引擎继续前进。

直到引擎开始从…‸tonight！处开始匹配，情况又变得有趣了。这一次，多选分支「night」终于可以匹配字符串的结尾部分了（于是整体匹配成功，现在引擎可以报告匹配成功了）。

回溯的两个要点

Two Important Points on Backtracking

回溯机制的基本原理并不难理解，还是有些细节对实际应用很重要。它们是，面对众多选择时，哪个分支应当首先选择？回溯进行时，应该选取哪个保存的状态？第一个问题的答案是下面这条重要原则：

> 如果需要在"进行尝试"和"跳过尝试"之间选择，对于匹配优先量词，引擎会优先选择"进行尝试"，而对于忽略优先量词，会选择"跳过尝试"。

此原则影响深远。对于新手来说，它有助于解释为什么匹配优先的量词是"匹配优先"的，但还不完整。要想彻底弄清楚这一点，我们需要了解回溯时使用的是哪个（或者是哪些个）之前保存的分支，答案是：

> 距离当前最近储存的选项就是当本地失败强制回溯时返回的。使用的原则是 LIFO（last in first out，后进先出）。

用面包屑比喻就很好理解——如果前面是死路，你只需要沿原路返回，直到找到一堆面包屑为止。你会遇到的第一堆面包屑就是最近洒下的。传统的 LIFO 比喻也是这样：就像堆叠盘子一样，最后叠上去的盘子肯定是最先拿下来的。

备用状态

Saved States

用 NFA 正则表达式的术语来说，那些面包屑相当于"备用状态（saved state）"。它们用来标记：在需要的时候，匹配可以从这里重新开始尝试。它们保存了两个位置：正则表达式中的位置，和未尝试的分支在字符串中的位置。因为它是 NFA 匹配的基础，我们需要再看一遍某些已经出现过的简单但详细的例子，说明这些状态的意义。如果你觉得现有的内容都不难懂，请继续阅读。

未进行回溯的匹配

来看个简单的例子，用「ab?c」匹配 abc。「a」匹配之后，匹配的**当前状态**如下：

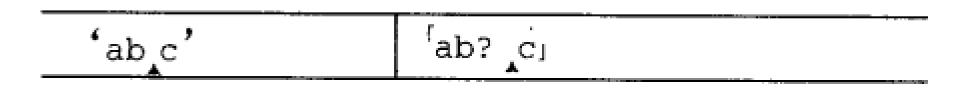
‘a▲bc’	「a▲b?c」

现在轮到「b?」了，正则引擎需要决定：是需要尝试「b」呢，还是跳过？因为?是匹配优先的，它会尝试匹配。但是，为了确保在这个尝试最终失败之后能够恢复，引擎会把：

‘a▲bc’	「ab?▲c」

添加到备用状态序列中。也就是说，稍后引擎可以从下面的位置继续匹配：从正则表达式中的「b?」之后，字符串的 b 之前（也就是当前的位置）匹配。这实际上就是跳过「b」的匹配，而问号容许这样做。

引擎放下面包屑之后，就会继续向前，检查「b」。在示例文本中，它能够匹配，所以新的当前状态变为：



‘ab▲c’	「ab?▲c」

最终的「c」也能成功匹配，所以整个匹配完成。备用状态不再需要了，所以不再保存它们。

进行了回溯的匹配

如果需要匹配的文本是‘ac’，在尝试「b」之前，一切都与之前的过程相同。显然，这次「b」无法匹配。也就是说，对「…?」进行尝试的路走不通。因为有一个备用状态，这个“局部匹配失败”并不会导致整体匹配失败。引擎会进行回溯，也就是说，把“当前状态”切换为最近保存的状态。在本例中，情况就是：

‘a▲c’	「ab?▲c」

在「b」尝试之前保存的尚未尝试的选项。这时候，「c」可以匹配 c，所以整个匹配宣告完成。

不成功的匹配

现在，我们用同样的表达式匹配‘abX’。在尝试「b」以前，因为存在问号，保存了这个备用状态：

‘a▲bX’	「ab?▲c」

「b」能够匹配，但这条路往下却走不通了，因为「c」无法匹配 X。于是引擎会回溯到之前的状态，“交还” b 给「c」来匹配。显然，这次测试也失败了。如果还有其他保存的状态，回溯会继续进行，但是此时不存在其他状态，在字符串中当前位置开始的整个匹配也就宣告失败。

事情到此结束了吗？没有。传动装置会继续“在字符串中前行，再次尝试正则表达式”，这可能被想象为一个伪回溯（pseudo-backtrack）。匹配重新开始于：



从这里重新开始整个匹配，如同之前一样，所有的道路都走不通。接下来的两次（从 ab‸X 到 abX‸）都告失败，所以最终会报告匹配失败。

忽略优先的匹配

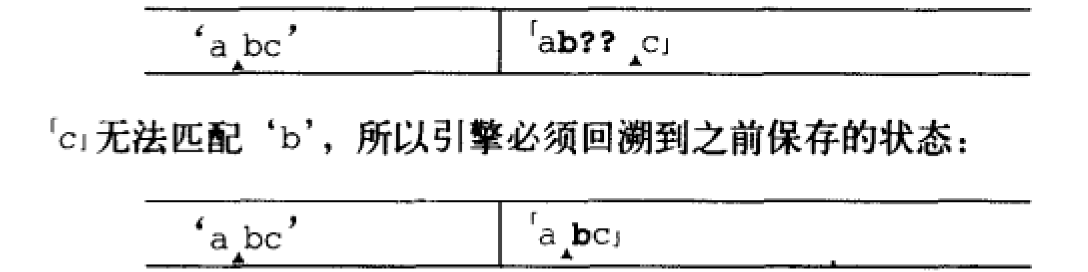
现在来看最开始的例子，使用忽略优先匹配量词，用「a**b??**c」来匹配 ‘abc’。「a」匹配之后的状态如下：

‘a‸bc’	「a‸**b??**c」

接下来轮到「b??」，引擎需要进行选择：尝试匹配「b」，还是忽略？因为??是忽略优先的，它会首先尝试忽略，但是，为了能够从失败的分支中恢复，引擎会保存下面的状态：

‘a‸bc’	「a‸**b**c」

到备用状态列表中。于是，引擎稍后能够用正则表达式中的「b」来尝试匹配文本中的 b（我们知道这能够匹配，但是正则引擎不知道，它甚至都不知道是否会要用到这个备用状态）。状态保存之后，它会继续向前，沿着忽略匹配的路走下去：



‘a‸bc’	「a**b??**‸c」

「c」无法匹配 ‘b’，所以引擎必须回溯到之前保存的状态：

‘a‸bc’	「a‸**b**c」

显然，此时匹配可以成功，接下来的「c」匹配 ‘c’。于是我们得到了与使用匹配优先的「a**b?**c」同样的结果，虽然两者所走的路不相同。

回溯与匹配优先

Backtracking and Greediness

如果工具软件使用的是NFA正则表达式主导的回溯引擎，理解正则表达式的回溯原理就成了高效完成任务的关键。我们已经看到「?」的匹配优先和「??」的忽略优先是如何工作的，现在来看星号和加号。

星号、加号及其回溯

如果认为「x*」基本等同于「x?x?x?x?x?x?…」（或者更确切地说是「(x(x(x(x…?)?)?)?)?)」）（注5），那么情况与之前没有大的差别。每次测试星号作用的元素之前，引擎都会保存一个状态，这样，如果测试失败（或者测试进行下去遭遇失败），还能够从保存的状态开始匹配。这个过程会不断重复，直到包含星号的尝试完全失败为止。

所以，如果用「[0-9]+」来匹配'a·1234·num'，「[0-9]」遇到4之后的空格无法匹配，而此时加号能够回溯的位置对应了四个保存的状态：

```
a 1▴234 num
a 12▴34 num
a 123▴4 num
a 1234▴ num
```

也就是说，在每个位置，「[0-9]」的尝试都代表一种可能。在「[0-9]」遇到空格匹配失败时，引擎回溯到最近保存的状态（也就是最下面的位置），选择正则表达式中的「[0-9]+▴」和文本中的'a·1234▴·num'。当然，到此整个正则表达式已经结束，所以我们知道，整个匹配宣告完成。

请注意，'a·▴1234·num'并不在列表中，因为加号限定的元素至少要匹配一次，这是必要条件。那么，如果正则表达式是「[0-9]*」，这个状态会保存吗？（提示：这个问题得动点脑筋）。要知道答案，◆请翻到下一页。

重新审视更完整的例子

有了更详细的了解之后，我们再来看看第152页的「^.*([0-9][0-9])」的例子。这一次，我们不是只用"匹配优先"来解释为什么会得到那样的匹配结果，我们能够根据NFA的匹配机制做出精确解释。

以'CA·95472·USA'为例。在「.*」成功匹配到字符串的末尾时，星号约束的点号匹配了13

注5：作为比较，请记住DFA并不关心表达式的形式，对DFA来说，这3个例子是无差别的。

个字符，同时保存了许多备用状态。这些状态表明稍后的匹配开始的位置：在正则表达式中是「^.*▴([0-9][0-9])」，在字符串中则是点号每次匹配时保存的备用状态。

现在我们已经到了字符串的末尾，并把控制权交给第一个「[0-9]」，显然这里的匹配不能成功。没问题，我们可以选择一个保存的状态来进行尝试（实际上保存了许多的状态）。现在回溯开始，把当前状态设置为最近保存的状态，也就是「.*」匹配最后的 A 之前的状态。忽略（或者，如果你愿意，可以使用“交还”）这个匹配，于是有机会用「[0-9]」匹配这个 A，但这同样会失败。

这种“回溯-尝试”的过程会不断循环，直到引擎交还 2 为止，在这里，第一个「[0-9]」可以匹配。但是第二个「[0-9]」仍然无法匹配，所以必须继续回溯。现在，之前尝试中第一个「[0-9]」是否匹配与本次尝试并无关系了，回溯机制会把当前的状态中正则表达式内的对应位置设置到第一个「[0-9]」以前。我们看到，当前的回溯同样会把字符串中的位置设置到 7 以前，所以第一个「[0-9]」可以匹配，而第二个「[0-9]」也可以（匹配 2）。所以，我们得到一个匹配结果‘CA·95472,·USA’，$1 得到 72。

需要注意的是：第一，回溯机制不但需要重新计算正则表达式和文本的对应位置，也需要维护括号内的子表达式所匹配文本的状态。在匹配过程中，每次回溯都把当前状态中正则表达式的对应位置指向括号之前，也就是「^.*▴([0-9][0-9])」。在这个简单的例子中，所以它等价于「^.*▴[0-9][0-9]」，因此我说“使用第一个[0-9]之前的位置”。然而，回溯对括号的这种处理，不但需要同时维护$1 的状态，也会影响匹配的效率。

最后需要注意的一点也许读者早就了解：由星号（或其他任何匹配优先量词）限定的部分**不受后面元素影响**，而只是匹配尽可能多的内容。在我们的例子中，「.*」在点号匹配失败之前，完全不知道，到底应该在哪个数字或者是其他什么地方停下来。在「^.*([0-9]+)」的例子中我们看到，「[0-9]+」只能匹配一位数字（☞153）。

关于匹配优先和回溯的更多内容

More About Greediness and Backtracking

NFA 和 DFA 都具备许多匹配优先相关的特性（也从中获益）。（DFA 不支持忽略优先，所以我们现在只关注匹配优先）。我将考察两种引擎共同支持的匹配优先特性，但只会用 NFA 来讲解。这些内容对 DFA 也适用，但原因与 NFA 不同。DFA 是匹配优先的，使用起来很

测试答案

◆ 162 页测试的答案

如果用「[0-9]*」匹配‘a·1234·num’，备用状态是否包括‘a·▴1234·num’？

答案是否定的。之所以提这个问题，是因为这种错误很常见。记得吗，由星号限定的部分总是能够匹配。如果整个表达式都由星号控制，它就能够匹配任何内容。在字符串的开始位置，传动机构对引擎进行第一次尝试时的状态，当然算匹配成功。在这种情况下，正则表达式匹配‘▴a·1234·num’，而且在此处结尾——它根本没有触及到那些数字。

如果没答对也不要紧，因为这种情况还是有可能发生的。如果在表达式中，「[0-9]*」之后还出现了某些元素，因为它们的存在，引擎在达到下面状态之前无法获得全局匹配：

‘a·▴1234…’	「▴[0-9]*…」

那么，尝试‘1’会生成下面的状态：

‘a·▴1234…’	「[0-9]* ▴…」

方便，这一点我们只需要记住就够了，而且介绍起来也很乏味。相反，NFA 的匹配优先就很值得一谈，因为 NFA 是表达式主导的，所以匹配优先的特性能产生许多神奇的结果。除了忽略优先，NFA 还提供了其他许多特性，比如环视、条件判断（conditions）、反向引用，以及固化分组。最重要的是 NFA 能让使用者直接操控匹配的过程，如果运用得当，这会带来很大的方便，但也可能带来某些性能问题（具体见第 6 章）。

不考虑这些差异的话，匹配的结果通常是相通的。我介绍的内容适用于两种引擎，除了使用表达式主导的 NFA 引擎。读完本章，读者会明白，在什么情况下两种引擎的结果会不一样，以及为什么会不一致。

匹配优先的问题

Problems of Greediness

在上例中已经看到，「.*」经常会匹配到一行文本的末尾（注 6）。这是因为「.*」匹配时只从自身出发，匹配尽可能多的内容，只有在全局匹配需要的情况下才会“被迫”交还一些字符。

注 6：如果工具软件或匹配模式设置了点号通配模式，「.*」能够匹配包含多行数据的字符串，包含所有的逻辑行，直到整个字符串的末尾。

有些时候问题很严重。来看用一个匹配双引号文本的例子。读者首先想到的可能是「".*"」，那么，请运用我们刚刚学到的关于「.*」的知识，猜一猜用它匹配下面文本的结果：

```
The name "McDonald's" is said "makudonarudo" in Japanese.
```

既然知道了匹配原理，其实我们并不需要猜测，因为我们"知道"结果。在最开始的双引号匹配之后，「.*」能够匹配任何字符，所以它会一直匹配到字符串的末尾。为了让最后的双引号能够匹配，「.*」会不断交还字符（或者，更确切地说，是正则引擎强迫它回退），直到满足为止。最后，这个正则表达式匹配的结果就是：

```
The name "McDonald's" is said "makudonarudo" in Japanese.
```

这显然不是我们期望的结果。我之所以提醒读者不要过分依赖「.*」，这就是原因之一，如果不注意匹配优先的特性，结果往往出乎意料。

那么，我们如何能够只取得"McDonald's"呢？关键的问题在于要认识到，我们希望匹配的不是双引号之间的"任何文本"，而是"除双引号以外的任何文本"。如果用「[^"]*」取代「.*」，就不会出现上面的问题。

使用「"[^"]*"」时，正则引擎的基本匹配过程跟上面是一样的。第一个双引号匹配之后，「[^"]*」会匹配尽可能多的字符。在本例中，就是 McDonald's，因为「[^"]」无法匹配之后的双引号。此时，控制权转移到正则表达式末尾的「"」。而它刚好能够匹配，所以获得全局匹配：

```
The name "McDonald's" is said "makudonarudo" in Japanese.
```

事实上，可能还存在一种出乎读者预料的情况，因为在大多数流派中，「[^"]」能够匹配换行符，而点号则不能。如果不想让表达式匹配换行符，可以使用「[^"\n]」。

多字符"引文"

Multi-Character "Quotes"

第1章出现了对HTML tag的匹配，例如，浏览器会把<B>very</B>中的"very"渲染为粗体。对<B>…</B>的匹配尝试看起来与对双引号匹配的情形很相似，只是"双引号"在这里成了多字符构成的<B>和</B>。与双引号字符串的例子一样，使用「.*」匹配多字符"引文"也会出错：

```
…<B>Billions</B> and <B>Zillions</B> of suns…
```

「<B>.*</B>」中匹配优先的「.*」会一直匹配该行结尾的字符，回溯只会进行到「</B>」能够匹配为止，也就是最后一个</B>，而不是与匹配开头的「<B>」对应的「</B>」。

不幸的是，因为结束 tag 不是单个字符，所以不能用之前的办法，也就是“排除型字符组”来解决，即我们不能期望用「<B>**[^</B>]***</B>」解决问题。字符组只能代表单个字符，而我们需要的「</B>」是一组字符。请不要被「[^</B>]」迷惑，它只是表示一个不等于<、>、/、B的字符，等价于「[^/<>B]」，而这显然不是我们想要的“除</B>结构之外的任何内容”（当然，如果使用环视功能，我们可以规定，在某个位置不应该匹配「</B>」，下一节会出现这种应用）。

使用忽略优先量词

Using Lazy Quantifiers

上面的问题之所以会出现，原因在于标准量词是匹配优先的。某些 NFA 支持忽略优先的量词（☞141），*? 就是与 * 对应的忽略优先量词。我们用「<B>**.*?**</B>」来匹配：

…<B>Billions</B> and <B>Zillions</B> of suns…

开始的「<B>」匹配之后，「.*?」首先决定不需要匹配任何字符，因为它是忽略优先的。于是，控制权交给后面的「<」符号：

‘…<B>▲Billions…’	「<B>.*?▲</B>」

此时「<」无法匹配，所以控制权交还给「.*?」，因为还有未尝试过的匹配可能（事实上能够进行多次匹配尝试）。它的匹配尝试是步步为营的（begrudgingly），先用点号来匹配…<B>B̲illions…中带下画线的 B。此时，*?又必须选择，是继续尝试匹配，还是忽略？因为它是忽略优先的，会首先选择忽略。接下来的「<」仍然无法匹配，所以「.*?」必须继续尝试未匹配的分支。在这个过程重复 8 次之后，「.*?」最终匹配了 Billions，此时，解下来的「<」（以及整个「</B>」）都能匹配：

…<B>Billions</B> and <B>Zillions</B> of suns…

所以我们知道了，星号之类的匹配优先量词有时候的确很方便，但有时候也会带来大麻烦。这时候，忽略优先的量词就能派上用场了，因为它们做到其他办法很难做到（甚至无法做到）的事情。当然，缺乏经验的程序员也可能错误地使用忽略优先量词。事实上，我们刚刚做的或许就不正确。如果用「<B>.*?</B>」来匹配：

…<B>Billions and <B>Zillions</B> of suns…

结果如上图，虽然我假设匹配的结果取决于具体的需求，但我仍然认为，这种情况下的结果不是用户期望的。不过，「.*?」必然会匹配 Zillions 左边的<B>，一直到</B>。

这个例子很好地说明了，为什么通常情况下，忽略优先量词并不是排除类的完美替身。在「".*"」的例子中，使用「[^"]」替换点号能避免跨越引号的匹配——这正是我们希望实现的功能。

如果支持**排除环视**（☞133），我们就能得到与排除型字符组相当的结果。比如，「(?!<B>)」这个测试，只有当<B>不在字符串中的当前位置时才能成功。这也是「<B>.*?</B>」中的点号期望匹配的内容，所以把点号改为「((?!<B>).)」得到的正则表达式，就能准确匹配我们期望的内容。把这些综合起来，结果有点儿难看懂，所以我选用带注释的宽松排列模式（☞111）来讲解：

```
<B>                        #  匹配开头的<B>
(                          #  然后只匹配尽可能少的内容
   (?! <B> )               #  如果不是<B>...
   .                       #  ... 任何字符都可以
)*?                        #
</B>                       #  ... 直到遇到结束标记
```

使用了环视功能之后，我们可以重新使用普通的匹配优先量词，这样看起来更清楚：

```
<B>                        #  匹配开头的<B>
(                          #  然后匹配尽可能多的内容
   (?! </? B>)             #  如果不是<B>，也不是</B> ...
   .                       #  ... 任何字符都可以
)*                         #  (现在是匹配优先的量词)
</B>                       #  <ANNO> ... 直到结束分隔符匹配
```

现在，环视功能会禁止正则表达式的主体（main body）匹配<B>和</B>之外的内容，这也是我们之前试图用忽略优先量词解决的问题，所以现在可以不用忽略优先功能了。这个表达式还有能够改进的地方，我们将在第 6 章关于效率的讨论中看到它（☞270）。

匹配优先和忽略优先都期望获得匹配

Greediness and Laziness Always Favor a Match

回忆一下第 2 章中显示价格的例子（☞51）。我们会在本章的多个地方仔细检查这个例子，所以我先重申一下基本的问题：因为浮点数的显示问题，“1.625”或者“3.00”有时候会变成“1.62500000002828”和“3.00000000028822”。为解决这个问题，我使用：

```
$price =~ s/(\.\d\d[1-9]?)\d*/$1/;
```

来保存$price小数点后头两到三位十进制数字。「\.\d\d」匹配最开始两位数字，而「[1-9]?」用来匹配可能出现的不等于零的第三个数字。

然后我写道：

> 到现在，我们能够匹配的任何内容都是希望保留的，所以用括号包围起来，作为$1 捕获。然后就能在 replacement 字符串中使用$1。如果这个正则表达式能够匹配的文本就等于$1，那么我们就是用一个文本取代它本身——这没有多少意义。然而，我们继续匹配$1 括号之外的部分。在替代字符串中它们并不出现，所以结果就是它们被删掉了。在本例中，“要删掉”的文本就是任何多余的数字，也就是正则表达式中末尾的「\d*」匹配的部分。

到现在看起来一切正常，但是，如果$price 的数据本身规范格式，会出现什么问题呢？如果$price 是 27.625，那么「\.\d\d[1-9]?」能够匹配整个小数部分。因为「\d*」无法匹配任何字符，整个替换就是用‘.625’替换‘.625’——相当于白费工夫。

结果当然是我们需要的，但是否存在更有效率的办法，只进行真正必要的替换（也就是，只有当「\d*」确实能够匹配到某些字符的时候才替换）呢？当然，我们知道怎样表示“至少一位数字”！只要把「\d*」替换为「\d+」就好了：

```
$price =~ s/(\.\d\d[1-9]?)\d+/$1/
```

对于“1.62500000002828”这样复杂的数字，正则表达式仍然有效；但是对于“9.43”这种数字，末尾的「\d+」不能匹配，所以不会替换。所以，这是个了不起的改动，对吗？不！如果要处理的数字是 27.625，情况如何呢？我们希望这个数字能够被忽略，但是这不会发生。请仔细想想 27.625 的匹配过程，尤其是表达式如何处理‘5’这个数字。

知道答案之后，这个问题就变得非常简单了。在「(\.\d\d[1-9]?)\d+」匹配 27.625 之后，「\d+」无法匹配。但这并不会影响整个表达式的匹配，因为就正则表达式而言，由「[1-9]」匹配‘5’只是**可选的**分支之一，还有一个备用状态尚未尝试。它容许「[1-9]?」匹配一个空字符，而把 5 留给至少必须匹配一个字符的「\d+」。于是，我们得到的是错误的结果：.625 被替换成了 .62。

如果「[1-9]?」是忽略优先的又如何呢？我们会得到同样的匹配结果，但不会有“先匹配 5 再回溯”的过程，因为忽略优先的「[1-9]??」首先会忽略尝试匹配的选项。所以，忽略优先量词并不能解决这个问题。

匹配优先、忽略优先和回溯的要旨

The Essence of Greediness, Laziness, and Backtracking

之前的章节告诉我们，正则表达式中的某个元素，无论是匹配优先，还是忽略优先，都是为全局匹配服务的，在这一点上（对前面的例子来说）它们没有区别。如果全局匹配需要，无论是匹配优先（或是忽略优先），在遇到“本地匹配失败”时，引擎都会回归到备用状态

（按照足迹返回找到面包屑），然后尝试尚未尝试的路径。无论匹配优先还是忽略优先，**只要引擎报告匹配失败，它就必然尝试了所有的可能。**

测试路径的先后顺序，在匹配优先和忽略优先的情况下是不同的（这就是两者存在的理由），但是，只有在测试完所有可能的路径之后，才会最终报告匹配失败。

相反，如果只有一条可能的路径，那么使用匹配优先量词和忽略优先量词的正则表达式都能找到这个结果，只是他们尝试路径的次序不同。在这种情况下，选择匹配优先还是忽略优先，并不会影响匹配的结果，受影响的只是引擎在达到最终匹配之前需要尝试的次数（这是关于效率的问题，第 6 章将会谈到）。

最后，如果存在不止一个可能的匹配结果，那么理解匹配优先、忽略优先和回溯的读者，就明白应该如何选择。「".*"」有 3 个可能的匹配结果：

```
The name "McDonald's" s said "makudonarudo" in Japenese.
```

我们知道，使用匹配优先星号的「".*"」匹配最长的结果，而使用忽略优先星号的「".*?"」匹配最短的结果。

占有优先量词和固化分组

Possessive Quantifiers and Atomic Grouping

上一页中‘.625’的例子展示了关于 NFA 匹配的重要知识，也让我们认识到，针对这个具体的例子，考虑不仔细就会带来问题。针对某些流派提供了工具来帮助我们，但是在接触它们以前，必须彻底弄明白“匹配优先、忽略优先和回溯的要旨”这一节。如果读者还有不明白的地方，请务必仔细阅读上一节。

那么，仍然来考虑‘.625’的例子，想想我们真正的目的。我们知道，如果匹配能够进行到「(\.\d\d[1-9]?)▲\d+」中标记的位置，我们就不希望进行回溯。也就是说，我们希望的是，如果可能，「[1-9]」能够一个字符，果真如此的话，我们不希望交还这个字符。或者说的更直白一些就是，如果需要的话，我们希望在「[1-9]」匹配的字符交还之前，整个表达式就匹配失败。（这个正则表达式匹配‘.625’时的问题在于，它**不会匹配**失败，而是进行回溯，尝试其他备用状态）。

那么，如果我们能够避免这些备用状态呢（也就是在[1-9]进行尝试之前，放弃「?」保存的状态，）？如果没有退路，「[1-9]」的匹配就不会交还。而这就是我们需要的！但是，如果没有了这个备用状态会发生什么？如果我们用这个表达式来匹配‘.5000’呢？此时「[1-9]」

不能匹配，就**确实**需要回溯，放弃「[1-9]」，让之后的「\d+」能够匹配需要删除的数字。

听起来，我们有两种相反的要求，但是仔细考虑考虑就会发现，我们真正需要的是，只有在某个可选元素已经匹配成功的情况下才抛弃此元素的“忽略”状态。也就是说，如果「[1-9]」能够成功匹配，就必须抛弃对应的备用状态，这样就永远也不会回退。如果正则表达式的流派支持固化分组「(?>…)」(☞139)，或者占有优先量词「[1-9]?+」(☞142)，就可以这么做。首先来看固化分组。

用(?>…)实现固化分组

具体来说，使用「(?>…)」的匹配与正常的匹配并无差别，但是如果匹配进行到此结构之后（也就是，进行到闭括号之后），那么此结构体中的所有备用状态都会被放弃。也就是说，在固化分组匹配结束时，它已经匹配的文本已经固化为一个单元，只能作为整体而保留或放弃。括号内的子表达式中未尝试过的备用状态都不复存在了，所以回溯永远也不能选择其中的状态（至少是，当此结构匹配完成时，“锁定（locked in)”在其中的状态)。

所以，让我们来看「(\.\d\d(?>[1-9]?))\d+」。在固化分组内，量词能够正常工作，所以如果「[1-9]」不能匹配，正则表达式会返回「?」留下的备用状态。然后匹配脱离固化分组，继续前进到「\d+」。在这种情况下，当控制权离开固化分组时，没有备用状态需要放弃（因为在固化分组中没有创建任何备用状态)。

如果「[1-9]」能够匹配，匹配脱离固化分组之后，「?」保存的备用状态仍然存在。但是，因为它属于已经结束的固化分组，所以会被抛弃。匹配‘.625’或者‘.625000’时就会发生这种情况。在后一种情况下，放弃那些状态不会带来任何麻烦，因为「\d+」匹配的是‘.625000’，到这里正则表达式已经完成匹配。但是对于‘.625’来说，因为「\d+」无法匹配，正则引擎需要回溯，但回溯又无法进行，因为备用状态已经不存在了。既然没有能够回溯的备用状态，整体匹配也就失败，‘.625’不需要处理，而这正是我们期望的。

固化分组的要旨

从第 168 页开始的“匹配优先、忽略优先和回溯的要旨”这一节，说明了一个重要的事实，即匹配优先和忽略优先都不会影响需要检测路径的本身，而只会影响检测的**顺序**。如果不能匹配，无论是按照匹配优先还是忽略优先的顺序，最终每条路径都会被测试。

然而，固化分组与它们截然不同，因为固化分组会放弃某些可能的路径。根据具体情况的不同，放弃备用状态可能会导致不同的结果：

- **毫无影响** 如果在使用备用状态之前能够完成匹配，固化分组就不会影响匹配。我们刚刚看过‘.625000’的匹配。全局匹配在备用状态尚未起作用之前就已经完成。
- **导致匹配失败** 放弃备用状态可能意味着，本来有可能成功的匹配现在不可能成功了。‘6.25’的例子就是如此。
- **改变匹配结果** 在某些情况下，放弃备用状态可能得到不同的匹配结果。
- **加快报告匹配失败的速度** 如果不能找到匹配对象，放弃备用状态可能能让正则引擎更快地报告匹配失败。先做个小测验，然后我们来谈这点。

❖ 小测验：「(?>.*?)」是什么意思？它能匹配什么文本？请翻到下页查看答案。

某些状态可能保留 在匹配过程中，引擎退出固化分组时，放弃的只是固化分组中创建的状态。而之前创建的状态依然保留，所以，如果后来的回溯要求退回到之前的备用状态，固化分组部分匹配的文本会全部交还。

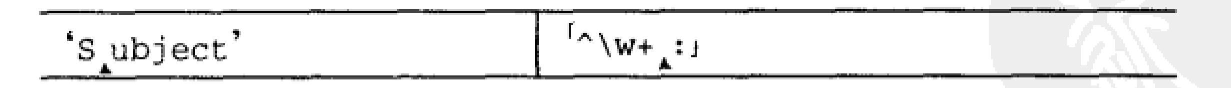
使用固化分组加快匹配失败的速度 我们一眼就能看出，「^\w+:」无法匹配‘Subject’，因为‘Subject’中不含冒号，但正则引擎必须经过尝试才能得到这个结论。

第一次检查「:」时，「\w+」已经匹配到了字符串的结尾，保存了若干状态——「\w」匹配的每个字符，都对应有“忽略”的备用状态（第一个除外，因为加号要求至少匹配一次）。「:」无法匹配字符串的末尾，所以正则引擎会回溯到最近的备用状态：



‘Subjec▴t’	「^\w+▴:」

此处的字符是‘t’，「:」仍然无法匹配。于是回溯-测试-失败的循环会不断发生，最终退回开始的状态：



‘S▴ubject’	「^\w+▴:」

此处仍然无法匹配成功，所以报告整个表达式无法匹配。

测验答案

❖ 171页测试的答案

「(?>.*?)」会匹配什么？

它永远无法匹配任何字符。充其量它只能算是个相当复杂的正则表达式，但不匹配任何字符。「.*?」是「.*」的忽略优先表示，它限定的是一个点号，所以首选的分支就是忽略点号，把匹配点号的状态保留下来备用。但是，这个保存的状态马上又会被放弃，因为匹配退出了固化分组，所以真正尝试的只有忽略点号的分支。总是被忽略的东西，实际上相当于不存在。

我们只消看一眼就能知道，所有的回溯都是白费工夫。如果冒号无法匹配最后的字符，那么它当然无法匹配「+」交还的任何字符。

既然我们知道，「\w+」匹配结束之后，从任何备用状态开始测试都不能得到全局匹配，就可以命令正则引擎不必检查它们：「^(?>\w+)」。我们已经全面了解了正则表达式的匹配过程，可以使用固化分组来控制「\w+」的匹配，放弃备用的状态（因为我们知道它们没有用），提高效率。如果存在可以匹配的文本，那么固化分组不会有任何影响，但是如果不存在能够匹配的文本，放弃这些无用的状态会让正则引擎更快地得出无法匹配的结论（先进的实现也许能够自动进行这样的优化，☞251）。

我们将在第6章看到（第269页），固化分组非常有价值，我怀疑它可能会成为最常用的技巧。

占有优先量词，?+、*+、++和{m,n}+

*Possessive Quantifiers, ?+, *+, ++, and {m,n}+*

占有优先量词与匹配优先量词很相似，只是它们从来不交还已经匹配的字符。我们在「^\w+」的例子中看到，加号在匹配结束之前创建了很少的备用状态，而占有优先的加号会直接放弃这些状态（或者，更形象地说，并不会创造这些状态）。

你也许会想，占有优先量词和固化分组关系非常紧密。像「\w++」这样的占有优先量词与「(?>\w+)」的匹配结果完全相同，只是写起来更加方便而已（注7）。使用占有优先量词，「^(?>\w+):」可以写作「^\w++:」，「(\.\d\d(?>[1-9]?))\d+」写做「(\.\d\d[1-9]?+)^\d+」。

注7：聪明的实现方式在处理占有优先量词时会更高效一些，☞250。

请务必区分「(?>M)+」和「(?M+)」。前者放弃「M」创建的备用状态，因为「M」不会制造任何状态，所以这样做没什么价值。而后者放弃「M+」创造的未使用状态，这样做显然有意义。

比较「(?>M)+」和「(?>M+)」，显然后者就对应于「M++」，但如果表达式很复杂，例如「(\\"|[^"])*+」，从占有优先量词转换为固化分组时大家往往会想到在括号中添加‘?>’得到「(?>\\"|[^"])*」。这个表达式或许有机会实现你的目的，但它显然不等于那个使用占有优先量词的表达式；它就好像是把「M++」写作「(?>M)+」一样。正确的办法是，去掉表示占有优先的加号，用固化分组把余下的部分包括起来：「(?>(\\"|[^"])*)」。

环视中的回溯

The Backtracking of Lookaround

初看时并不容易认识到，环视（见第 2 章，☞59）与固化分组和占有优先量词有紧密的联系。环视分为 4 种：肯定型（positive）和否定型（negative），顺序环视与逆序环视。它们只是简单地测试，其中表达式能否在当前位置匹配后面的文本（顺序环视），或者前面的文本（逆序环视）。

深入点看，在 NFA 的世界中包含了备用状态和回溯，环视是怎么实现的？环视结构中的子表达式有自己的世界。它也会保存备用状态，进行必要的回溯。如果整个子表达式能够成功匹配，结果如何呢？**肯定**型环视会认为自己匹配成功；而**否定**环视会认为匹配失败。在任何一种情况下，因为关注的只是匹配存在与否（在刚才的例子中，的确存在匹配），此匹配尝试所在的“世界”，包括在尝试中创造的所有备用状态，都会被放弃。

如果环视中的子表达式无法匹配，结果如何呢？因为它只应用到这个“世界”中，所以回溯时只能选择当前环视结构中的备用状态。也就是说，如果正则表达式发现，需要进一步回溯到当前的环视结构的起点以前，它就认为当前的子表达式无法匹配。对肯定型环视来说，这就意味着失败，而对于否定型环视来说，这意味着成功。在任何一种情况下，都没有保留备用状态（如果有，那么子表达式的匹配尝试就没有结束），自然也谈不上放弃。

所以我们知道，只要环视结构的匹配尝试结束，它就不会留下任何备用状态。任何备用状态和例子中肯定环视成功时的情况一样，都会被放弃。我们在其他什么地方看到过放弃状态？当然是固化分组和占有优先量词。

用肯定环视模拟固化分组

如果流派本身支持固化分组，这么做可能有点多此一举，但如果流派不支持固化分组，这么做就很有用：如果所使用的工具支持肯定环视，同时可以在肯定环视中使用捕获括号（大多数风格都支持，但也有些不支持，Tcl 就是如此），就能模拟实现固化分组和占有优先量词。「(?>*regex*)」可以用「(?=(*regex*))\1」来模拟。举例来说，比较「(?>\w+):」和「^(?=(\w+))\1:」。

环视中的「\w+」是匹配优先的，它会匹配尽可能多的字符，也就是整个单词。因为它在环视结构中，当环视结束之后，备用状态都会放弃（和固化分组一样）。但与固化分组不一样的是，虽然此时确实捕获了这个单词，但它不是全局匹配的一部分（这就是环视的意义）。这里的关键就是，后面的「\1」捕获的就是环视结构捕获的单词，而这当然会匹配成功。在这里使用「\1」并非多此一举，而是为了把匹配从这个单词结束的位置进行下去。

这个技巧比真正的固化分组要慢一些，因为需要额外的时间来重新匹配「\1」的文本。不过，因为环视结构可以放弃备用状态，如果冒号无法匹配，它的失败会来得更快一些。

多选结构也是匹配优先的吗

Is Alternation Greedy?

多选分支的工作原理非常重要，因为在不同的正则引擎中它们是迥然不同的。如果遇到的多个多选分支都能够匹配，究竟会选择哪一个呢？或者说，如果不只一个多选分支能够匹配，最后究竟应该选择哪一个呢？如果选择的是匹配文本最长的多选分支，有人也许会说多选结构也是匹配优先的；如果选择的是匹配文本最短的多选分支，有人也许会说它是忽略优先的？那么（如果只能是一个的话）究竟是哪个？

让我们看看 Perl、PHP、Java、.NET 以及其他语言使用的传统型 NFA 引擎。遇到多选结构时，这种引擎会按照从左到右的顺序检查表达式中的多选分支。拿正则表达式「^(Subject|Date):·」来说，遇到「Subject|Date」时，首先尝试的是「Subject」。如果能够匹配，就转而处理接下来的部分（也就是后面的「:·」）。如果无法匹配，而此时又有其他多选分支（就是例子中的「Date」），正则引擎会回溯来尝试它。这个例子同样说明，**正则引擎会回溯到存在尚未尝试的多选分支的地方**。这个过程会不断重复，直到完成全局匹配，或者所有的分支（也就是本例中的所有多选分支）都尝试穷尽为止。

所以，对于常见的传统型 NFA 引擎，用「tour|to|tournament」来匹配 'three·tournaments·won' 时，会得到什么结果呢？在尝试到 'three·▲tournaments·won' 时，在每个位置进行的匹配尝试都会失败，而且每次尝试时，都会检查所有的多选分支（并且失败）。而在这个位置，第一个多选分支「tour」能够匹配。因为这个多选结构是正则表达式中的最后部分，「tour」匹配结束也就意味着整个表达式匹配完成。其他的多选分支就不会尝试了。

因此我们知道，多选结构既不是匹配优先的，也不是忽略优先的，而是按顺序排列的，至少对传统型 NFA 来说是如此。这比匹配优先的多选结构更有用，因为这样我们能够对匹配的过程进行更多的控制——正则表达式的使用者可以用它下令："先试这个，再试那个，最后试另一个，直到试出结果为止"。

不过，也不是所有的流派都支持按序排列的多选结构。DFA 和 POSIX NFA 确实有匹配优先的多选结构，它们总是匹配所有多选分支中能匹配最多文本的那个（也就是本例中的「tournament」）。但是，如果你使用的是 Perl、PHP、.NET、java.util.regex，或者其他使用传统型 NFA 的工具，多选结构就是**按序排列**的。

发掘有序多选结构的价值

Taking Advantage of Ordered Alternation

回过头来看第 167 页「(\.\d\d[1-9]?)\d*」的例子。如果我们明白，「\.\d\d[1-9]?」其实等于「\.\d\d」或者「\.\d\d[1-9]」，我们就可以把整个表达式重新写作「(\.\d\d|\.\d\d[1-9])\d*」。（这并非必须的改动，只是举例说明）。这个表达式与之前的完全一样吗？如果多选结构是匹配优先的，那么答案就是肯定的，但如果多选结构是有序的，两者就完全不一样。

我们来看多选结构有序的情形。首先选择和测试的是第一个多选分支，如果能够匹配，控制权就转移到紧接的「\d*」那里。如果还有其他的数字，「\d*」能够匹配它们，也就是任何不为零的数字，它们是原来问题的根源（如果读者还记得，当时的问题就在于，这位数字我们只希望在括号里匹配，而不通过括号外面的「\d*」）。所以，如果第一个多选分支无法匹配，第二个多选分支同样无法匹配，因为二者的开头是一样的。即使第一个多选结构无法匹配，正则引擎仍然会对第二个多选分支进行徒劳的尝试。

不过，如果交换多选分支的顺序，变成「(\.\d\d[1-9]|\.\d\d)\d*」，它就等价于匹配优先的「(\.\d\d[1-9]?)\d*」。如果第一个多选分支结尾的「[1-9]」匹配失败，第二个多选分支仍然有机会成功。我们使用的仍然是有序排列的多选结构，但是通过变换顺序，实现了匹配优先的功能。

第一次拆分「[1-9]?」成两个多选分支时，我们把较短的分支放在了前面，得到了一个不具备匹配优先功能的「?」。在这个具体的例子中，这么做没有意义，因为如果第一个多选分支不能匹配，第二个肯定也无法匹配。我经常看到这样没有意义的多选结构，对传统型 NFA 来说，这肯定不对。我曾看到有一本书以「a((ab)*|b*)」为例讲解传统型 NFA 正则表达式的括号。这个例子显然没有意义，因为第一个多选分支「(ab)*」永远也不会匹配失败，所以后面的其他多选分支毫无意义。你可以继续添加：

「a*((ab)*|b*|.*|partridge·in·a·pear·tree|[a-z])」

这个正则表达式的意义都不会有丝毫的改变。要记住的是，如果多选分支是有序的，而能够匹配同样文本的多选分支又不只一个，就要小心安排多选分支的先后顺序。

有序多选结构的陷阱

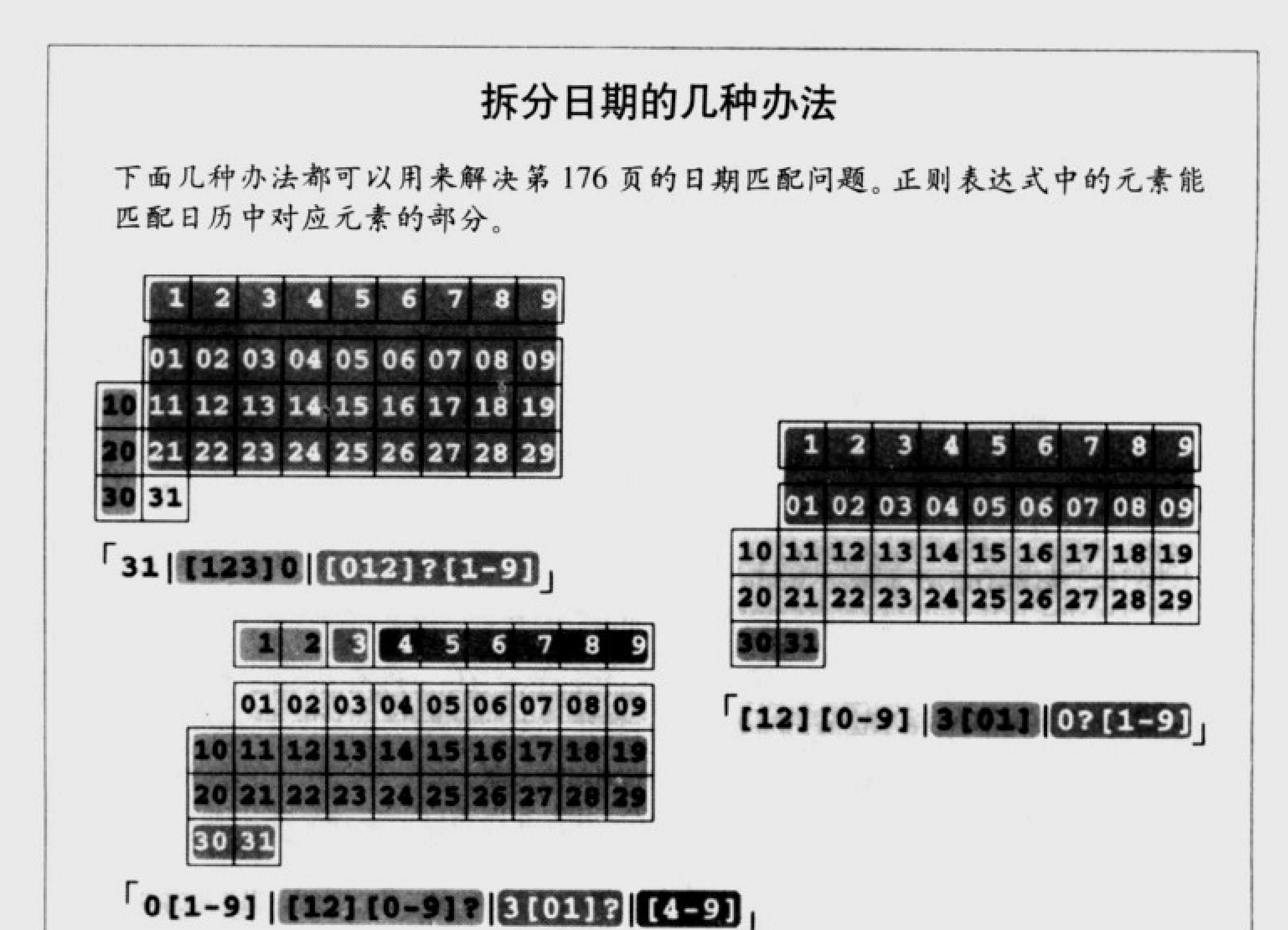
有序多选分支容许使用者控制期望的匹配，因此极为便利，但也会给不明就里的人造成麻烦。如果需要匹配‘Jan 31’之类的日期，我们需要的就不是简单的「Jan·[0123][0-9]」，因为这样的表达式能够匹配‘Jan·00’和‘Jan·39’这样的日期，却无法匹配‘Jan·7’。

一种办法是把日期部分拆开。用「0?[1-9]」匹配可能以 0 开头的前九天的日期。用「[12][0-9]」处理十号到二十九号，用「3[01]」处理最后两天。把上面这些连起来，就是「Jan·(0?[1-9]|[12][0-9]|3[01])」。

如果用这个表达式来匹配‘Jan 31 is Dad's birthday’，结果如何呢？我们希望获得的当然是‘Jan 31’，但是有序多选分支只会捕获‘Jan 3’。奇怪吗？在匹配第一个多选分支「0?[1-9]」时，前面的「0?」无法匹配，但是这个多选分支仍然能够匹配成功，因为「[1-9]」能够匹配‘3’。因为此多选分支位于正则表达式的末尾，所以匹配到此完成。

如果我们重新安排多选结构的顺序环视，把能够匹配的数字最短的放到最后，这个问题就解决了：「Jan·([12][0-9]|3[01]|0?[1-9])」。

另一种办法是使用「Jan·(31|[123]0|[012]?[1-9])」。但这也要求我们仔细地安排多选分支的顺序避免问题。还有一种办法是「Jan·(0[1-9]|[12][0-9]?|3[01]?|[4-9])」，这样不论顺序环视如何都能获得正确结果。比较和分析这 3 个不同的表达式，会有很多发现（我会给读者一些时间来想这个问题，尽管下一页的补充内容会有所帮助）。



NFA、DFA 和 POSIX

NFA, DFA, and POSIX

最左最长规则

"The Longest-Leftmost"

之前我们说过：如果传动装置在文本的某个特定位置启动 DFA 引擎，而在此位置又有一个或多个匹配的可能，DFA 就会选择这些可能中最长的。因为在所有同样从最左边开始的可能的匹配文本中它是最长的，所以叫它“最左最长的（longest-leftmost）”匹配。

绝对最长

这里说的“最长”不限于多选结构。看看 NFA 如何用「one(self)?(selfsufficient)?」来匹配字符串 oneselfsufficient。NFA 首先匹配「one」，然后是匹配优先的「(self)?」，留下「(selfsufficient)?」来匹配 sufficient。它显然无法匹配，但整个表达式并不会因此匹配失败，因为这个元素不是必须匹配的。所以，传统型 NFA 返回 oneself sufficient，放弃没有尝试的状态（POSIX NFA 的情况与此不同，我们稍后将会看到）。

与此相反，DFA 会返回更长的结果：oneselfsufficient。如果最开始的「(self)?」因为某些原因无法匹配，NFA 也会返回跟 DFA 一样的结果，因为「(self)?」无法匹配，「(selfsufficient)?」就能成功匹配。传统型 NFA 不会这样，但是 DFA 则会这样，因为会选择最长的可能匹配。DFA 同时记录多个匹配，在任何时候都清楚所有的匹配可能，所以它能做到这一点。

我选这个简单的例子是因为它很容易理解，但是我希望读者能够明白，这个问题在现实中很重要。举例来说，如果希望匹配**连续多行文本**，常见的情况是，一个逻辑行（logical line）可以分为许多现实的行，每一行以反斜线结尾，例如：

```
SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c \
        missing.c msg.c node.c re.c version.c
```

读者可能希望用「^\w+=.*」来匹配这种“var=value”的数据，但是正则表达式无法识别连续的行（在这里我们假设点号无法匹配换行符）。为了匹配多行，读者可能需要在表达式最后添加「(\\\n.*)*」，得到「^\w+=.*(\\\n.*)*」。显然，这意味着任何后继的逻辑行都能匹配，只要他们以反斜线结尾。这看起来没错，但在传统型 NFA 中行不通。「.*」到达行尾的时候，**已经匹配了反斜线**，而表达式中后面的部分不会强迫进行回溯（☞152）。但是，DFA 能够匹配更长的多行文本，因为它确实是最长的。

如果能够使用忽略优先的量词，也许可以考虑用它们来解决问题，例如「^\w+=.*?(\\\n.*?)*」。这样点号每次实际匹配任何字符之前，都需要测试转义的换行符部分，这样「\\」就能够匹配换行符之前的反斜线。不过这也行不通。如果忽略优先量词匹配某些可选的部分，必然是在全局匹配必须的情况下发生。但是在本例中，「=」后面的所有部分都不是必须匹配的，所以没有东西会强迫忽略优先量词匹配任何字符。忽略优先的正则表达式只能匹配‘SRC=’，这显然不是我们期望的结果。

这个问题还有其他的解决办法，我们会在下一章继续这个问题（☞186）。

POSIX 和最左最长规则

POSIX and the Longest-Leftmost Rule

POSIX 标准规定，如果在字符串的某个位置存在多个可能的匹配，应当返回的是最长的匹配。

POSIX 标准文档中使用了“最左边开始的最长匹配（longest of the leftmost）”。它并没有规

定必须使用 DFA，那么，如果希望使用 NFA 来实践 POSIX，程序员应该如何做？如果你希望执行 POSIX NFA，那么必须找到完整的 oneselfsufficient 和所有的连续行，虽然这个结果是违反 NFA“天性”的。

传统型 NFA 引擎会在第一次找到匹配时停下来，但是如果让它继续尝试其他分支（状态）会怎样呢？每次匹配到表达式的末尾时，它都会获得另一个可能的匹配结果。如果所有的分支都穷尽了，就能从中选择最长的匹配结果。这样，我们就得到了一台 POSIX NFA。

在上面的例子中，NFA 匹配「(self)?」时保存了一个备用状态：「one(self)? ▴(selfsufficient)?」在「one▴selfsufficient」。传统型 NFA 在 oneselfsufficient 之后停止匹配，而 POSIX NFA 仍然会继续检查余下的所有状态，最终得到那个更长的结果（其实是最长的）oneselfsufficient。

第 7 章有一个例子，可以让 Perl 模拟 POSIX 的做法，返回最长的匹配字符（☞225）。

速度和效率

Speed and Efficiency

如果传统型 NFA 的效率是我们应当关注的问题（对提供回溯的传统型 NFA 来说，这确实是一个问题），那么 POSIX NFA 的效率就更值得关注，因为它需要进行更多的回溯。POSIX NFA 需要尝试正则表达式的所有变体（译注 4）。第 6 章告诉我们，正则表达式写得糟糕的话，匹配的效率就会很低。

DFA 的效率

文本主导的 DFA 巧妙地避免了回溯造成的效率问题。DFA 同时记录了所有可能的匹配，这样来提高速度。它是如何做到这一切的呢？

DFA 引擎需要更多的时间和内存，它第一次遇见正则表达式时，在做出任何尝试之前会用比 NFA 详细得多的（也是截然不同的）办法来分析这个正则表达式。开始尝试匹配的时候，它已经内建了一张路线图（map），描述“遇到这个和这个字符，就该选择这个和那个可能的匹配”。字符串中的每个字符都会按照这张路线图来匹配。

有时候，构造这张路线图可能需要相当的时间和内存，不过只要建立了针对特定正则表达式的路线图，结果就可以应用到任意长度的文本。这就好像为你的电动车充电一样。首先，你得把车停到车库里面，插上电源等待一段时间，但只要发动了汽车，清洁的能源就会源源而来。

译注 4：permutation，指一个正则表达式能够匹配的各种形式的文本。

NFA 理论与现实

NFA（译注 5）真正的数学和计算学意义是不同于通常说的“NFA 引擎”的。在理论上，NFA 和 DFA 引擎应该匹配完全一样的文本，提供完全一样的功能。但是在实际中，因为人们需要更强的功能，更具表达能力的正则表达式，它的语意发生了变化。反向引用就是一例。

DFA 引擎的设计方案就排除了反向引用，但是对于真正（数学意义上的）的 NFA 引擎来说，提供反向引用的支持只需要很小的改动。这样我们就得到了一个功能更强大的工具，但它绝对是**非正则**（nonregular）的（数学意义上的）。这是什么意思呢？或许，你不应该继续叫它“NFA”，而是“不正则表达式（nonregular expressions）”，因为这个名词才能描述（在数学意义上）新的情形。但是实际没人这么做，所以这个名字就这样流传下来，虽然实现方式都不再是（数学意义上的）NFA。

这对用户有什么意义？显然没有什么意义。作为用户，你不需要关心它是正则还是非正则，而只需要知道它能为你做什么（也就是本章的内容）。

对那些希望了解更多的正则表达式的理论的人，经典的计算机科学教学文本是，Aho、Sethi、Ullman 的 *Compilers—Principles, Techniques, and Tools*（Addison-Wesley, 1986）的第 3 章，通常说的“恐龙书”，因为它的封面。更确切地说，这是一条“红龙”，“绿龙”指的是它的前任，Aho 和 Ullman 的 *Principles of Compiler Design*.

小结：NFA 与 DFA 的比较

Summary: NFA and DFA in Comparison

NFA 与 DFA 各有利弊。

DFA 与 NFA：在预编译阶段（pre-use compile）的区别

在使用正则表达式搜索之前，两种引擎都会编译表达式，得到一套内化形式，适应各自的匹配算法。NFA 的编译过程通常要快一些，需要的内存也更少一些。传统型 NFA 和 POSIX NFA 之间并没有实质的差别。

译注 5：此处 NFA 指的是非确定型有穷自动机。

DFA 与 NFA：匹配速度的差别

对于“正常”情况下的简单文本匹配测试，两种引擎的速度差不多。一般来说，DFA 的速度与正则表达式无关，而 NFA 中两者直接相关。

传统的 NFA 在报告无法匹配以前，必须尝试正则表达式的所有变体。这就是为什么我要用整章（第 6 章）来论述提高 NFA 表达式匹配速度的技巧。我们将会看到，有时候一个 NFA 永远无法结束匹配。传统型 NFA 如果能找到一个匹配，肯定会停止匹配。

相反，POSIX NFA 必须尝试正则表达式的所有变体，确保获得最长的匹配文本，所以如果匹配失败，它所花的时间与传统型 NFA 一样（有可能很长）。因此，对 POSIX NFA 来说，表达式的效率问题更为重要。

在某种意义上，我说得绝对了一点，因为优化措施通常能够减少获得匹配结果的时间。我们已经看到，优化引擎不会在字符串开头之外的任何地方尝试带「^」锚点的表达式，我们会在第 6 章看到更多的优化措施。

DFA 不需要做太多的优化，因为它的匹配速度很快，不过最重要的是，DFA 在预编译阶段所作的工作提供的优化效果，要好于大多数 NFA 引擎复杂的优化措施。

现代 DFA 引擎经常会尝试在匹配需要时再进行预编译，减少所需的时间和内存。因为应用的文本各异，通常情况下大部分的预编译都是白费工夫。因此，如果在匹配过程确实需要的情况下再进行编译，有时候能节省相当的时间和内存（技术术语就是“延迟求值（lazy evaluation）”）。这样，正则表达式、待匹配的文本和匹配速度之间就建立了某种联系。

DFA 与 NFA：匹配结果的差别

DFA（或者 POSIX NFA）返回最左边的最长的匹配文本。传统型 NFA 可能返回同样的结果，当然也可能是别的文本。针对某一型具体的引擎，同样的正则表达式，同样的文本，总是得到同样的结果，在这个意义上来说，它不是“随机”的，但是其他 NFA 引擎可能返回不一样的结果。事实上，我见过的所有传统型 NFA 返回的结果都是一样的，但并没有任何标准来硬性规定。

DFA 与 NFA：能力的差异

NFA 引擎能提供一些 DFA 不支持的功能，例如：

- 捕获由括号内的子表达式匹配的文本。相关的功能是反向引用和后匹配信息（after-match information），它们描述匹配的文本中每个括号内的子表达式所匹配文本的位置。
- 环视，以及其他复杂的零长度确认（注 8）（☞133）。
- 非匹配优先的量词，以及有序的多选结构。DFA 很容易就能支持选择最短的匹配文本（尽管因为某些原因，这个选项似乎从未向用户提供过），但是它无法实现我们讨论过的局部的忽略优先性和有序的多选结构。
- 占有优先量词（☞142）和固化分组（☞139）。

兼具 DFA 的速度和 NFA 的功能：正则表达式的终极境界

我已经多次说过，DFA 不能支持捕获括号和反向引用。这无疑是对的，但这并不是说，我们不能组合不同的技术，以达到正则表达式的终极境界。180 页的补充内容描述了 NFA 为了追求更强大的功能，如何脱离了纯理论的道路和限制，DFA 的情况也是如此。受自身结构的限制，DFA 进行这种突破更加困难，但并非不可能。

GNU *grep* 采取了一种简单但有效的策略。它尽可能多地使用 DFA，在需要反向引用的时候，才切换到 NFA。GNU awk 的办法也差不多——在进行“是否匹配”的检查时，它采用 GNU *grep* 的 DFA 引擎，如果需要知道具体的匹配文本的内容，就采用不同的引擎。这里的“不同的引擎”就是 NFA，利用自己的 gensub 函数，GNU awk 能够很方便地提供捕获括号。

Tcl 的正则引擎由 Henry Spencer（你或许记得，这个人在正则表达式的早期发展和流行中扮演了重要的角色）开发，它也是混合型的。Tcl 引擎有时候像 NFA——它支持环视、捕获括号、反向引用和忽略优先量词。但是，它也确实能提供 POSIX 的最左最长匹配（☞177），但没有我们将在第 6 章看到的 NFA 的问题。这点确实很棒。

注 8：*lex* 提供的 *trailing context* 等价于正则表达式末尾的零长度肯定环视，但它并不能应用到末尾之外的环视结构中。

DFA 与 NFA：实现难度的差异

尽管存在限制，但简单的 DFA 和 NFA 引擎都很容易理解和实现。对效率（包括时间和空间效率）和增强性能的追求，令实现越来越复杂。

用代码长度来衡量的话，支持 NFA 正则表达式的 *ed* Version 7（1979 年 1 月发布）只有不到 350 行的 C 代码（所以，整个 *grep* 只有区区 478 行代码）。Henry Spencer1986 年免费提供的 Version 8 正则程序差不多有 1 900 行 C 代码，1992 年 Tom Lord 的 POSIX NFA package `rx`（被 GNU sed 和其他工具采用）长达 9 700 行。

为了糅合 DFA 和 NFA 的优点，GNU *egrep* Version 2.4.2 使用了两个功能完整的引擎（差不多 8 900 行代码），Tcl 的 DFA/NFA 混合引擎（请看上一页的补充内容）更是长达 9 500 行。

某些实现很简单，但这并不是说它们支持的功能有限。我曾经想要用 Pascal 的正则表达式来处理某些文本。从毕业以后我就没用过 Pascal 了，但是写个简单的 NFA 引擎并不需要太多工夫。它并不追求花哨，也不追求速度，但是提供了相对全面的功能，非常实用。

总结

Summary

如果你希望一遍就能读懂本章的所有内容，大概得做点准备。至少，这些东西不那么容易理解。我花了些时间才理解它，花了更长的时间才真正弄懂。我希望这章简要的讲解能够降低读者理解的难度。我尝试过简单地解释，同时不要调入太简单的陷阱（不幸的是，太过直白的解释总是妨碍了真正的理解）。本章有许多这样的陷阱，所以我在其中安排了许多对其他页的引用，在下面的摘要中，读者可以很快地找到其他的内容。

实现正则表达式匹配引擎有两种常见的技术，一种是“表达式主导的 NFA”（☞153），另一种是“文本主导的 DFA”（☞155）。它们的全称见第 156 页。

这两种技术，结合POSIX标准，可以按照实用标准划分3种正则引擎：

- 传统型NFA（汽油驱动，功能强大）。
- POSIX NFA（汽油驱动，符合标准）。
- DFA（不一定符合POSIX）（电力驱动，运转稳定）。

为了对手头的工具有个大致的了解，你需要知道它采用的是什么引擎，以对正则表达式做相应的调校。最常见的引擎就是传统型NFA，其次是DFA。表4-1（☞145）列出了若干常用工具及它们使用的引擎类型，“测试引擎的类型”（☞146）给出了测试引擎类型的方法。

对于任何引擎来说，都有一条通用的规则：开始位置靠先的匹配文本优先于开始位置靠后的匹配文本。因为“传动机构”会从前往后在文本的各个位置展开尝试（☞148）。

对于从某个位置开始的匹配：

文本主导的DFA引擎：

寻找可能的最长的匹配文本。不再介绍（☞177）。稳定、速度快（☞179），讲解起来很麻烦。

表达式主导的NFA引擎：

匹配过程中可能需要“反复尝试（work through）”。NFA匹配的灵魂是**回溯**（☞157，162）。控制匹配过程的元字符：标准量词（星号等等）是匹配优先的（☞151），其他量词是忽略优先或者占有优先的（☞169）。在传统型NFA中，多选结构是有序排列的（☞174），在POSIX NFA中是匹配优先的。

POSIX NFA 必须找到最长的匹配文本。但是，匹配并不难理解，只须考虑效率（第6章的问题）。

传统型 NFA 控制能力最强的正则引擎，因此使用者可以使用该引擎的表达式主导性质来精确控制匹配过程。

理解本章的概念和练习是书写正确而高效的正则表达式的基础，这也是接下来两章的主题。

第5章 正则表达式实用技巧

Practical Regex Techniques

现在我们已经掌握了编写正则表达式所需的基本知识，我希望在更复杂的环境中应用这些知识来处理更复杂的问题。每个正则表达式都必须在下面两个方面求得平衡：准确匹配期望匹配的内容，忽略不期望匹配的字符。我们已经看过许多例子都说明，如果应用得当，匹配优先非常有用，但如果不够小心，也可能带来麻烦，在本章我们还将看到许多例子。

NFA 引擎还需要平衡另外一个因素：效率，这也是下一章的主题。设计糟糕的正则表达式——即使可以认为没犯错误——也足以让引擎瘫痪。

本章出现的主要是各种实例，我会带领读者循着我的思路去解决各种问题。某些例子或许对读者并没有现实价值，但我仍然推荐读者阅读这些实例。

例如，即使你的工作不涉及 HTML，我仍推荐你从处理 HTML 的实例中吸取知识。原因在于，编写巧妙的正则表达式不仅仅是一种手艺（skill）——而且还是一种艺术（art）。它的教授和学习，不是依靠罗列规则，而是依靠经验，所以，我用这些例子告诉读者，自己在过去的若干年从经验中获得的深刻启示。

当然，读者仍然需要自己掌握这些知识，但是研究本章的例子是个好的起点。



正则表达式的平衡法则

Regex Balancing Act

好的正则表达式必须在这些方面求得平衡：

- 只匹配期望的文本，排除不期望的文本。
- 必须易于控制和理解。
- 如果使用 NFA 引擎，必须保证效率（如果能够匹配，必须很快地返回匹配结果，如果不能匹配，应该在尽可能短的时间内报告匹配失败）。

这些方面常常是与具体文本相关的。如果我只使用命令行，只需要快速地 *grep* 某些东西，可能不会过分关心匹配的准确性，通常也不会花太多精力来调校。我不在乎多花点时间来手工排查，因为我能够迅速地在输出中找到自己需要的内容。但是，如果处理重要的程序，就需要花费时间精力来保证正确性：如果需要，正则表达式也可能很复杂。这些因素都需要权衡。

即使使用同样的程序，效率也是与具体文本相关的。如果是 NFA，用「`^-(display|geometry|cemap|…|quick24|random|raw)$`」之类长长的正则表达式来检验命令行参数的效率就很低，因为多选分支过多，但如果它只用于检验命令行参数（可能只是在程序开始的时候运行若干次），即使所需时间比正常的长 100 倍也不要紧，因为这时候效率并不是问题。但是，如果要逐行检查很大的文件，低效率的程序运行起来会让你痛苦不堪。

若干简单的例子

A Few Short Examples

匹配连续行（续前）

Continuing with Continuation Lines

继续前一章中匹配连续行的例子（☞178），我们发现(在传统型 NFA 中使用「`^\w+=.*(\\\n.*)*`」并不能匹配下面的两行文本：

```
SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c\
        missing.c msg.c node.c re.c version.c
```

问题在于，第一个「`.*`」一直匹配到反斜线之后，这样「`(\\\n.*)*`」就不能按照预期匹配反斜线了。所以，本章出现的第一条经验就是：如果不需要点号匹配反斜线，就应该在正则表达式中做这样的规定。我们可以把每个点号替换成「`[^\n\\]`」（请注意，`\n` 包含在排除性字符组中。你应该记得，原来的正则表达式的假设之一就是，点号不会匹配换行符，我们也不希望它的替代品能够匹配换行符☞119 页）。

于是，我们得到：

「^\w+=[^\n\\]*(\\\n[^\n\\]*)*」

它确实能够匹配连续行，但因此也产生了一个新的问题：这样反斜线就不能出现在一行的非结尾位置。如果需要匹配的文本中包含其他的反斜线，这个正则表达式就会出问题。现在我们假设它会包含，所以需要继续改进正则表达式。

迄今为止，我们的思路都是，“匹配一行，如果还有连续行，就继续匹配”。现在换另一种思路，这种思路我觉得通常都会奏效：集中关注在特定时刻真正容许匹配的字符。在匹配一行文本时，我们期望匹配的要么是普通（除反斜线和换行符之外）字符，要么是反斜线与其他任何字符的结合体。在点号通配模式中，「\\.」能匹配反斜线加换行符的结合体。

所以，正则表达式就变成了「^\w+=([^\n\\]|\\.)*」，在点号通配模式下。因为开头是「^」，如果需要，可能得使用增强的文本行锚点匹配模式（☞112）。

但是，这个答案仍然不够完美——我们会在下一章讲解效率问题时再次看到它（☞270）。

匹配 IP 地址

Matching an IP Address

来看个复杂点的例子，匹配一个 IP（Internet Protocol，因特网协议）地址：用点号分开的四个数，例如 1.2.3.4。通常情况下，每个数都有三位，例如 001.002.003.004。你可能会想到用「[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*」从文本中提取一个 IP 地址，但是这个表达式显然不够精致，它甚至会匹配‘and then ....?’。仔细看看就会发现，这个表达式甚至不需要匹配任何数字——它只需要三个点号（当然也可能包括其间的数字）。

为解决这个问题，我们首先把星号改成加号，因为我们知道，每一段必须有至少一位数字。为确保整个字符串的内容就是一个 IP 地址，我们可以在首尾加上「^...$」，于是我们得到：

「^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$」

如果用「\d」替换「[0-9]」，就得到「^\d+\.\d+\.\d+\.\d+$」，这样可能更好看一些（注 1），但是，这个表达式仍然会捕获一些并非 IP 地址的数据，例如‘1234.5678.9101112.131415’

注 1：这倒不一定，主要看读者的喜好。我发现，在复杂的正则表达式中，「\d」比「[0-9]」更容易理解，但是，在某些系统中，这二者并不等同。在支持 Unicode 的系统中，「\d」或许能匹配非 ASCII 的数字。

（IP 地址的每个字段都在 0-255 以内）。那么，你可以强行规定每个字段必须包含三位数字，就是「^\d\d\d\.\d\d\d\.\d\d\d\.\d\d\d$」，但这样未免太不灵活（too specific）了。即使某个字段只有一位或者两位数字（例如 1.234.5.67），也应该匹配。如果流派支持区间量词{*min,max*}，就可以这么写「^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$」。如果不支持，则可以用「\d\d?\d?」或者「\d(\d\d?)?」。这两种方式略有不同，但都能匹配一到三个数字。

现在，正则表达式中的匹配精度可能已经满足需求了。如果要更精确，就必须考虑到，「\d{1,3}」能够匹配 999，而它超过了 255，所以它不是一个合法的 IP 地址。

我们有好几种办法来匹配 0 和 255 之间的数字。最笨的办法就是「0|1|2|3|…253|254|255」。不过这又不能处理以 0 开头的数字，所以必须写成「0|00|000|1|01|001…」，这样一来，正则表达式就长得过分了。对于 DFA 引擎来说，问题还只是它太长太繁杂——但匹配的速度与其他等价正则表达式是一样的。但对于 NFA 引擎，太多的多选分支简直就是效率杀手。

实际的解决办法是，关注字段中什么位置可以出现哪些数字。如果一个字段只包含一个或者两个数字，就无需担心这个字段的值是否合法，所以「\d|\d\d」就能应付。也不比担心那些以 0 或者 1 开头的三位数，因为 000-199 都是合法的 IP 地址。所以我们加上「[01]\d\d」，得到「\d|\d\d|[01]\d\d」。你可能觉得这有点像第 1 章里匹配时间的例子（☞28），和前一章中匹配日期的例子（☞177）。

继续看这个正则表达式，以 2 开头的三位数字，如果小于 255 就是合法的，所以第二位数字小于 5 就代表整个数也是合法的。如果第二位数字是 5，第三位数字就必须小于 6。这可以表示为「2[0-4]\d|25[0-5]」。

现在这个正则表达式有点看不懂了，但分析之后还是能够理解其中包含的思路。结果就是「\d|\d\d|[01]\d\d|2[0-4]\d|25[0-5]」。其实我们可以合并前面三个多选分支，得到「[01]?\d\d?|2[0-4]\d|25[0-5]」。在 NFA 中，这样做的效率更高，因为任何多选分支匹配失败都会导致回溯。请注意，第一个多选分支中用的是「\d\d?」，而不是「\d?\d」，这样，如果根本不存在数字，NFA 会更快地报告匹配失败。我把这个问题的分析留给读者——通过一个简单的验证就能发现二者的区别。我们还可以做些修改进一步提高这个表达式的效率，不过这要留待下一章讨论了。

现在这个表达式能够匹配 0 到 255 之间的数，我们用括号把它包起来，用来取代之前表达式中的「\d{1,3}」，就得到：

「^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.
([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])$」

这可真叫复杂！需要这么麻烦吗？这得根据具体需求来决定。这个表达式只会匹配合法的 IP 地址，但是它也会匹配一些语意不正确的 IP 地址，例如 0.0.0.0（所有字段都为零的 IP 地址是非法的）。使用环视功能（☞133）可以在「^」后添加「(?!0+\.0+\.0+\.0+$)」，但是某些时候，处理各种极端情形会降低成本/收益的比例。某些情况下，更合适的做法就是不依赖正则表达式完成全部工作。例如，你可以只使用「^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$」，用括号把每个字段括起来，把数字变成程序中的$1、$2、$3、$4，这样就可以用其他程序来验证了。

确定应用场合（context）

这个正则表达式必须借助锚点「^」和「$」才能正常工作，认识到这一点很重要。否则，它就可能匹配 ip=72123.3.21.993，如果使用传统型 NFA，则可能匹配 ip=123.3.21.223。

在第二个例子中，这个表达式甚至连最后的 223 都无法完整匹配。但是，问题并**不在于**表达式本身，因为没有东西（例如分隔符，或者末尾的锚点）强迫它匹配 223。最后那个分组的第一个多选分支「[01]?\d\d?」，匹配了前面两位数字，如果末尾没有「$」，匹配到此就结束了。在前一章日期匹配的例子中，我们可以安排多选分支的次序来达到期望的目的。现在我们也把能把匹配三位数字的多选分支放在最前面，这样在匹配两位数的多选分支获得尝试机会之前，任何三位数都能完全匹配（DFA 和 POSIX NFA 当然不需要这样安排，因为它们总是返回最长的匹配文本）。

无论是否重新排序，第一个错误仍然不可避免。"啊哈！"，你可能会想，"我可以用单词分界符锚点来解决这个问题。"不幸的是，这也不能奏效，因为这样的正则表达式仍然能够匹配 1.2.3.4.5.6。为了避免匹配这样内嵌的文本，我们必须确保匹配文本两侧至少没有数字或者点号。如果使用环视，可以在原来表达式的首尾添加「(?<![\w.])…(?![\w.])」来保证匹配文本之前（以及之后）不出现「[\w.]」能匹配的字符。如果不支持环视，在首尾添加「(^|·)…(·|$)」也能够应付某些情况。

处理文件名

Working with Filenames

处理文件名和路径，例如 Unix 下的`/usr/local/bin/Perl`或者 Windows 下的`\Program Files\Yahoo!\Messenger`，很适合用来讲解正则表达式的应用。因为“动手（using）”比“观摩（reading）”更有意思，我会同时用 Perl、PHP（preg 程序）、Java 和 VB.NET 来讲解。如果你对其中的某些语言不感兴趣，不妨完全跳过那些代码——其中蕴含的思想才是最重要的。

去掉文件名开头的路径

第一个例子是去掉文件名开始的路径，例如把`/usr/local/bin/gcc`变成`gcc`。从本质层面来考虑问题是成功的一半。在本例中，我们希望去掉在最后的斜线（含）之前（在 Windows 中是反斜线）的任何字符。如果没有斜线最好，因为什么也不用干。我曾说过，「`.*`」常常被滥用，但是此处我们需要匹配优先的特性。「`^.*/`」中的「`.*`」可以匹配一整行，然后回退（也就是回溯）到最后的斜线，来完成匹配。

下面是四种语言的代码，去掉变量`f`中的文件名中开头的路径。对于 Unix 的文件名：

语　言	代　码
Perl	`$f =~ s{^.*/}{};`
PHP	`$f = preg_replace('{^.*/}', '', $f);`
`java.util.regex`	`f = f.replaceFirst("^.*/","");`
VB.NET	`f = Regex.Replace(f, "^.*/","");`

正则表达式（或者说用来表示正则表达式的字符串）以下画线标注，正则表达式相关的组件则由粗体标注。

下面是处理 Windows 文件名的代码，Windows 中的分隔符是反斜线而不是斜线，所以要用正则表达式「`^.*\\`」。在正则表达式中，我们需要在反斜线前再加一个反斜线，才能表示转义的反斜线，不过，在中间两段程序中添加的这个反斜线本身也需要转义：

语　言	代　码
Perl	`$f =~ s/^.*\\//;`
PHP	`$f = preg_replace('/^.*\\\/', '', $f);`
`java.util.regex`	`f = f.replaceFirst("^.*\\\\","");`
VB.NET	`f = Regex.Replace(f, "^.*\\","");`

从中很容易看出各种语言的差异，尤其是 Java 中那 4 个反斜线（☞101）。

有一点请务必记住：别忘了时常想想匹配失败的情形。在本例中，匹配失败意味着字符串中没有斜线，所以不会替换，字符串也不会变化，而这正是我们需要的。

为了保证效率，我们需要记住 NFA 引擎的工作原理。设想下面这种情况：我们忘记在正则表达式的开头添加「^」符号（这个符号很容易忘记），用来匹配一个恰好没有斜线的字符串。同样，正则引擎会在字符串的起始位置开始搜索。「.*」抵达字符串的末尾，但必须不断回退，以找到斜线或者反斜线。直到最后它交还了匹配的所有字符，仍然无法匹配。所以，正则引擎知道，**在字符串的起始位置**不存在匹配，但这远远没有结束。

接下来传动装置开始工作，从在目标字符串的第 2 个字符开始，依次尝试匹配整个正则表达式。事实上，它需要在字符串的每个位置（从理论上说）进行扫描-回溯。文件名通常很短，因此这不是一个问题，但原理确实如此。如果字符串很长，就可能存在大量的回溯（当然，DFA 不存在这个问题）。

在实践中，经过合理优化的传动装置能够认识到，对几乎所有以「.*」开头的正则表达式来说，如果在某个字符串的起始位置不能匹配，也就不能在其他任何位置匹配，所以它只会在字符串的起始位置（☞246）尝试一次。不过，在正则表达式中写明这一点更加明智，在例子中我们正是这样做的。

从路径中获取文件名

另一种办法是忽略路径，简单地匹配最后的文件名部分。最终的文件名就是从最后一个斜线开始的所有内容：「[^/]*$」。这一次，锚点不仅仅是一种优化措施，我们确实需要在结尾设置一个锚点。现在我们可以这样做，以 Perl 来说明：

```
$WholePath =~ m{([^/]*)$};     # 利用正则表达式检测$wholePath
$Filename = $1;                # 记录匹配内容
```

你也许注意到了，这里并没有检查这个正则表达式能否匹配，因为它总是能匹配。这个表达式的唯一要求就是，字符串有$能够匹配的结束位置，而即使是空字符串也有一个结束位置。因此，我用$1 来引用括号内的表达式匹配的文本，因为它必定包括某些字符（如果文件名以斜线结尾，结果就是空字符）。

这里还需要考虑到效率：在 NFA 中，「[^/]*$」的效率很低。仔细想想 NFA 引擎的匹配过程，你会明白它包括了太多的回溯。即使是短短的‘/usr/local/bin/perl’，在获得匹配结果

之前，也要进行四十多次回溯。考虑从…‸local…开始的尝试。「[^/]*」一直匹配到第二个 l，之后匹配失败，然后对 l、a、c、o、l 的存储状态依次尝试「$」(都无法匹配)。如果这还不够，又会从…l‸ocal/…开始重复这个过程，接着从…lo‸cal/…开始，不断重复。

这个例子不应该消耗我们太多的精力，因为文件名一般都很短（40 次回溯几乎可以忽略不计——4000 万次回溯才真正要紧）。再一次，重要的是理解问题本身，这样才能选择合适的通用规则来解决具体的问题。

需要指出的是，纵然本书是关于正则表达式的，但正则表达式也不总是最优解。例如，大多数程序设计语言都提供了处理文件名的非正则表达式函数。不过为了讲解正则表达式，我仍会继续下去。

所在路径和文件名

下一步是把完整的路径分为所在路径和文件名两部分。有许多办法做到这一点，这取决于我们的要求。开始，你可能想要用「^(.*)/(.*)$」的$1 和$2 来提取这两者。看起来这个正则表达式非常直观，但知道了匹配优先量词的工作原理之后，我们知道第一个「.*」会首先捕获所有的文本，而不给「/」和$2 留下任何字符。第一个「.*」能交还字符的唯一原因，就是在尝试匹配「/(.*)$」时进行的回溯。这会把“交还的”部分留给后面的「.*」。因此，$1 就是文件所在的路径，$2 就是文件的名字。

需要注意的是，我们依靠开头的「(.*)/」来确保第二个「(.*)」不会匹配任何斜线。理解匹配优先之后，我们知道这没问题。如果要做的更精确，可以使用「[^/]*」来捕捉文件名。于是我们得到「^(.*)/([^/]*)$」。这个表达式准确地表达了我们的意图，一眼就能看明白。

这个表达式有个问题，它要求字符串中必须出现一个斜线，如果我们用它来匹配 file.txt，因为无法匹配，所以没有结果。如果我们希望精益求精，可以这样：

```
if ($WholePath = ~m!^(.*)/([^/]*)$!) {
 #  能够匹配 --$1 和$2 都合法
 $LeadingPath = $1;
 $FileName = $2;
} else {
 #  无法匹配，文件名中不包含‘/’
 $LeadingPath = ".";  #  所以"file.txt"应该是"./file.txt" ("."表示当前路径)
 $FileName = $WholePath;
}
```

匹配对称的括号

Matching Balanced Sets of Parentheses

对称的圆括号、方括号之类的符号匹配起来非常麻烦。在处理配置文件和源代码时，经常需要匹配对称的括号。例如，解析 C 语言代码时可能需要处理某个函数的所有参数。函数的参数包含在函数名称之后的括号里，而这些参数本身又有可能包含嵌套的函数调用或是算式中的括号。我们先不考虑嵌套的括号，你或许会想到「\bfoo\([^)]*\)」，但这行不通。

秉承 C 的光荣传统，我把示范函数命名为 foo。表达式中的标记部分是用来捕获参数的。对于 foo(2,·4.0)和 foo(somevar,·3.7)之类的参数，这个表达式完全没问题。但是，它也可以匹配 foo(bar(somevar),·3.7)，这可不是我们需要的。所以要用到比「[^)]*」更聪明的办法。

为了匹配括号部分，我们可以尝试下面的这些正则表达式：

1. \(.*\)　　括号及括号内部的任何字符。
2. \([^)]*\)　　从一个开括号到最近的闭括号。
3. \([^()]*\)　　从一个开括号到最近的闭括号，但是不容许其中包含开括号。

图 5-1 显示了对一行简单代码应用这些表达式的结果。

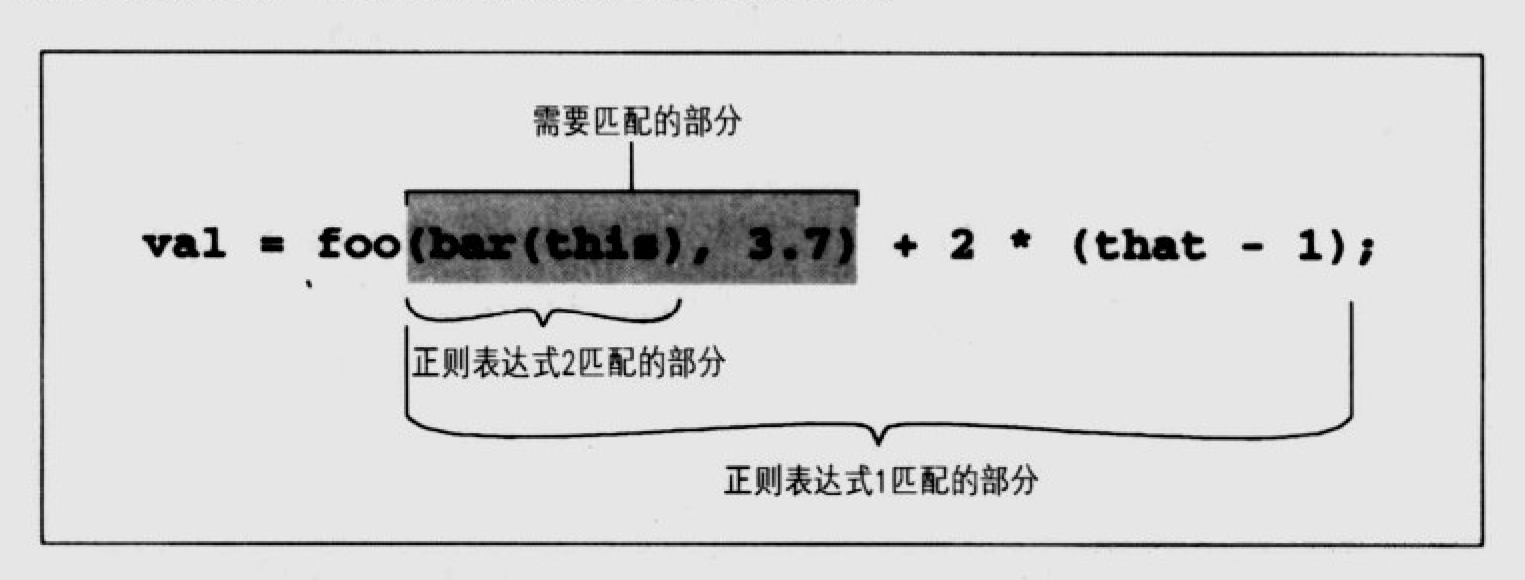


图 5-1：三个表达式的匹配位置

我们看到，第一个正则表达式匹配的内容太多（注 2），第二个正则表达式匹配的内容太少，第三个正则表达式无法匹配。孤立地看，第三个正则表达式能够匹配‘(this)’，但是因为表达式要求它必须紧接在 foo 之后，所以无法匹配。所以，这三个表达式都不合格。

注 2：「.*」很容易出问题，所以使用「.*」时必须格外谨慎，明确是否真的需要用一个星号来约束点号。有时候确实必须这么做，不过通常「.*」都不是合适的选择。

真正的问题在于，大多数系统中，**正则表达式无法匹配任意深度的嵌套结构**。在很长的时间内，这是放之四海而皆准的规则，但是现在 Perl、.NET 和 PCRE/PHP 都提供了解决的办法（参见第 328、436、475 页）。但是，即使不用这些功能，我们也可以用正则表达式来匹配**特定深度**的嵌套括号，但不是**任意深度**的嵌套括号。处理单层嵌套的正则表达式是：

```
「\[^()]*(\([^()]*\)[^()]*)*\)」
```

这样类推下去，更深层次的嵌套就复杂得可怕。但是，下面的 Perl 程序，在指定嵌套深度 $depth 之后，生成的正则表达式可以匹配最大深度为$depth 的嵌套括号。它使用的是 Perl 的"*string* x *count*"运算符，这个运算符会把 *string* 重复 *count* 次：

```
$regex = '\('.'(?:[^()]|\(' x $depth . '[^()]*' . '\))*' x $depth .'\)';
```

这个表达式留给读者分析。

防备不期望的匹配

Watching Out for Unwanted Matches

有个问题很容易忘记，即，如果待分析的文本不符合使用者的预期，会发生什么。假设你需要编写一个过滤程序，把普通文本转换为 HTML，你希望把一行连字符号转换为 HTML 中代表一条水平线的<HR>。如果使用搜索-替换命令 s/-*/<HR>/，它能替换期望替换的文本，但只限于它们在行开头的情况。很奇怪吗？事实上，s/-*/<HR>/会把<HR>添加到**每一行**的开头，而无论这些行是否以连字符开头。

请记住，如果某个元素的匹配没有硬性规定任何必须出现的字符，那么它总能匹配成功。「-*」从字符串的起始位置开始尝试匹配，它会匹配可能的任何连字符。但是，如果没有连字符，它仍然能匹配成功，这完全符合星号的定义。

在某位我非常尊重的作者的作品中出现过类似的例子，他用这个例子来讲解正则表达式匹配一个数，或者是整数或者是浮点数。在它的正则表达式中，这个数可能以负数符号开头，然后是任意多个数字，然后是可能的小数点，再是任何多的数字。他的正则表达式是「-?[0-9]*\.?[0-9]*」。

确实，这个正则表达式可以匹配 1、-272.37、129238843.、.191919，甚至是-.0 这样的数。这样看来，它的确是个不错的正则表达式。

但是，你想过这个表达式如何匹配'this·has·no·number''nothing·here'或是空字符串吗？仔细看看这个正则表达式——**每一个部分**都不是匹配必须的。**如果**存在一个数，**如果**正则表达式从在字符串的起始位置开始，的确能够匹配，但是因为**匹配没有任何必须元素**。

此正则表达式可以匹配每个例子中字符串开头的空字符。实际上它甚至可以匹配‘num·123’开头的空字符，因为这个空字符比数字出现得更早。

所以，把真正意图表达清楚是非常重要的。一个浮点数必须要有至少一位数字，否则就不是一个合法的值。我们首先假设，在小数点之前至少有一位数字（之后我们会去掉这个条件）。如果是，我们需要用加号来控制这些数字「-?[0-9]+」。

如果要用正则表达式来匹配可能存在的小数点（及其后的数字），就必须认识到，小数部分必须紧接在小数点之后。如果我们简单地用「\.?[0-9]*」，那么无论小数点是否存在，「[0-9]*」都可能匹配。

解决的办法还是厘清我们的意图：小数点（以及之后的数字）是可能出现的：「(\.[0-9]*)?」。这里，问号限定（也可以叫“统治 governs”或者“控制 controls”）的不再是小数点，而是小数点和后面的小数部分。在这个结合体内部，小数点是必须出现的，如果没有小数点，「[0-9]*」根本谈不上匹配。

把它们结合起来，就得到「-?[0-9]+(\.[0-9]*)?」。这个表达式不能匹配‘.007’，因为它要求整数部分必须有一位数字。如果我们作些修改，容许整数部分为空，就必须同时修改小数部分，否则这个表达式就可以**匹配空字符**（这是我们一开始就准备解决的问题）。

解决的办法是为无法覆盖的情况添加多选分支：「-?[0-9]+(\.[0-9]*)?|-?\.[0-9]+」。这样就能匹配以小数点开头的小数（小数点是必须的）。仔细看看，仔细看看。你注意到了吗？第二个多选分支同样能够匹配负数符号开头的小数？这很容易忘记。当然，你也可以把「-?」提出来，放到所有多选结构的外面：「-?([0-9]+(\.[0-9]*)?|\.[0-9]+)」。

虽然这个表达式比最开始的好得多，但它仍然会匹配‘2003.04.12’这样的数字。要想真正匹配期望的文本，同时忽略不期望的文本，求得平衡，就必须了解实际的待匹配文本。我们用来提取浮点数的正则表达式必须包含在一个大的正则表达式内部，例如用「^…$」或者「num\s*=\s*…$」。

匹配分隔符之内的文本

Matching Delimited Text

匹配用分隔符（以某些字符表示）之类的文本是常见的任务，之前的匹配双引号内的文本和 IP 地址只是这类问题中的两个典型例子。其他的例子还包括：

- 匹配‘/*’和‘*/’之间的 C 语言注释。
- 匹配一个 HTML tag，也就是尖括号之内的文本，例如`<CODE>`。
- 提取 HTML tag 标注的文本，例如在 HTML 代码‘`a <I>super exciting</I> offer!`’中的‘`super exciting`’。
- 匹配 *.mailrc* 文件中的一行内容。这个文件的每一行都按下面的数据格式来组织：

  alias 简称 电子邮件地址

  例如‘`alias jeff jfriedl@regex.info`’（在这里，分隔符是每个部分之间的空白和换行符）。
- 匹配引文字符串（quoted string），但是容许其中包含转义的引号，例如‘`a passport needs a "2\"x3\" likeness" of the holder`’。
- 解析 CSV（逗号分隔值，comma-separated values）文件。

总的来说，处理这些任务的步骤是：

1. 匹配起始分隔符（opening delimiter）。
2. 匹配正文（main text，即结束分隔符之前的所有文本）。
3. 匹配结束分隔符。

我曾经说过，如果结束分隔符不只一个字符，或者结束分隔符能够出现在正文中，这种任务就很难完成。

容许引文字符串中出现转义引号

来看 **2\"x3\"**的例子，这里的结束分隔符是一个引号，但正文也可能包含转义之后的引号。匹配开始和结束分隔符很容易，诀窍就在于，匹配正文的时候不要超越结束分隔符。

仔细想想正文里能够出现的字符，我们知道，如果一个字符不是引号，也就是说如果这个字符能由「`[^"]`」匹配，那么它肯定属于正文。不过，如果这个字符是一个引号，而它前面又有一个反斜线，那么这个引号也属于正文。把这个意思表达出来，使用环视（☞133）功能来处理“如果之前有反斜线”的情况，就得到「`"([^"]|(?<=\\)")*"`」，这个表达式完全能够匹配 **2\"x3\"**。

不过，这个例子也能用来说明，看起来正确的正则表达式如何会匹配意料之外的文本，它虽然看起来正确，但不是任何情况下都正确。我们希望它匹配下面这个无聊的例子中的划线部分：

```
Darth Symbol: "/-|-\\" or "[^-^]"
```

但它匹配的是：

```
Darth Symbol: "/-|-\\" or "[^-^]"
```

这是因为，第一个闭引号之前的确存在一个反斜线。但这个反斜线本身是被转义的，它不是用来转义之后的双引号的（也就是说这个引号其实是表示引用文本的结束）。而逆序环视无法识别这个被转义的反斜线，如果在这个引号之前有任意多个‘\\’，用逆序环视只会把事情弄得更糟。原来的表达式的真正问题在于，如果反斜线是用来转义引号的，在我们第一次处理它时，不会认为它是表示转义的反斜线。所以，我们得用别的办法来解决。

仔细想想我们想要匹配的位于开始分隔符和结束分隔符之间的文本，我们知道，其中可以包括转义的字符（「\\.」），也可以包括非引号的任何字符「[^"]」。于是我们得到「"(\\.|[^"])*"」。不错，现在这个问题解决了。不幸的是，这个表达式还有问题。不期望的匹配仍然会发生，比如对这个文本，它应该是无法匹配的，因为其中没有结束分隔符。

```
"You need a 2\"x3\" Photo.
```

为什么能匹配呢？回忆一下“匹配优先和忽略优先都期望获得匹配”（☞167）。即使这个表达式一开始匹配到了引号之后的文本，如果找不到结束的引号，它就会回溯，到达

‘…2x\"3▲\"•…’	「(\\.|▲[^"])」

从这里开始，「[^"]」匹配到反斜线，之后的那个引号被认为是一个结束的引号。

这个例子给我们的重要启示是：

> 如果回溯会导致不期望，与多选结构有关的匹配结果，问题很可能在于，任何成功的匹配都不过是多选分支的排列顺序造成的偶然结果。

实际上，如果我们把这个正则表达式的多选分支反过来排列，它就会错误地匹配**任何**包含转义双引号的字符串。真正的问题在于，各个多选分支能够匹配的内容发生了重叠。

那么，应该如何解决这个问题呢？就像第186页的那个连续行的例子一样，我们必须确保，这个反斜线不能以其他的方式匹配，也就是说把「[^"]」改为「[^\\"]」。这样就能识别双引

号和文本中的“特殊”反斜线，必须根据情况分别处理。结果就是「"(\\.|[^\\"])*"」，它工作得很好（尽管这个正则表达式能够正常工作，但对于NFA引擎来说，仍然有提升效率的改进，我们会在下一章更详细地看这个例子，☞222）。

这个例子告诉我们一条重要的原理：

> 不应该忘记考虑这样的“特殊”情形：例如针对“糟糕（bad）”的数据，正则表达式**不应该能够匹配**。

我们的修改是正确的，但是有意思的是，如果有占有优先量词（☞142）或者是固化分组（☞139），这个正则表达式可以重新写作「"(\\.|[^"])*+"」和「"(?>(\\.|[^"])*)"」。这两个正则表达式禁止引擎回溯到可能出问题的地方，所以它们都可以满足要求。

理解占有优先量词和固化分组解决此问题的原理非常有价值，但是我仍然要继续之前的修正，因为对读者来说它更具描述性（更直观）。其实在这个问题上，我也愿意使用占有优先量词和固化分组——不是为了解决之前的问题，而是为了效率，因为这样报告匹配失败的速度更快。

了解数据，做出假设

Knowing Your Data and Making Assumptions

现在是时候强调我曾经数次提到过的关于构建和使用正则表达式的一般规则了。知道正则表达式会在什么情况中应用，关于目标数据又有什么样的假设，这非常重要。即使简单如「a」这样的数据也假设目标数据使用的是作者预期的字符编码（☞105）。这都是一些很基本的常识，所以我一直没有过分细致地介绍。

但是，许多对某个人来说明显的常识，可能对其他人来说并不明显。例如，前一节的解决办法假设转义的换行符不会被匹配，或者会被应用于点号通配模式（☞111）。如果我们真的想要保证点号可以匹配换行符，同时流派也支持，我们应该使用「(?s:.)」。

前一节中我们还假设了正则表达式将应用的数据类型，它不能处理表示其他用途的双引号。如果用这个正则表达式来处理任何程序的源代码，就可能出错，因为注释中可能包括双引号。

对数据做出假设，对正则表达式的应用方式做出假设，都无可厚非。问题在于，假设如果

存在，通常会过分乐观，也会低估了作者的意图和正则表达式最终应用间的差异。记录下这些假设会有帮助。

去除文本首尾的空白字符

Stripping Leading and Trailing Whitespace

去除文本首尾的空白字符并不难做到，这是经常要完成的任务。总的来说最好的办法是使用下面两个替换：

```
s/^\s+//;
s/\s+$//;
```

为了增加效率，我们用「+」而不是「*」，因为如果事实上没有需要删除的空白字符，就不用做替换。

出于某些考虑，人们似乎更希望用一个正则表达式来解决整个问题，所以我会提供一些方法供比较。我不推荐这些办法，但对理解这些正则表达式的工作原理及其问题所在，非常有意义。

`s/\s*(.*?)\s*$/$1/s`

这个正则表达式曾被用作降解忽略优先量词的绝佳例子，但现在不是了，因为人们认识到它比普通的办法慢得多（在 Perl 中要慢 5 倍）。之所以效率这么低，是因为忽略优先约束的点号每次应用时都要检查「\s*$」。这需要大量的回溯。

`s/^\s*((?:.*\S)?)\s*$/$1/s`

这个表达式看起来比上一个要复杂，不过它的匹配倒是很容易理解，而且所花的时间也只是普通方法的 2 倍。在「^\s*」匹配了文本开头的空格之后，「.*」马上匹配到文本的末尾。后面的「\S」强迫它回溯直到找到一个非空的字符，把剩下的空白字符留给最后的「\s*$」，捕获括号之外的。

问号在这里是必须的，因为如果一行数据只包含空白字符的行，必须出现问号，表达式才能正常工作。如果没有问号，可能会无法匹配，错过这种只有空白字符的行。

`s/^\s+|\s+$//g`

这是最容易想到的正则表达式，但它不正确（其实这三个正则表达式都不正确），这种顶极的（top-leveled）多选分支排列严重影响本来可能使用的优化措施（参见下一章）。

/g 这个修饰符是必须的，它容许每个多选分支匹配，去掉开始和结束的空格。看起来，用/g 是多此一举，因为我们知道我们只希望去掉最多两部分空白字符，每部分对应单独的子表达式。这个正则表达式所用的时间是简单办法的 4 倍。

测试时我提到了相对速度，但是实际的相对速度取决于所用的软件和数据。例如，如果目标文本非常非常长，而且在首尾只有很少的空格，中间的那个表达式甚至会比简单的方法更快。不过，我自己在程序中仍然使用下面两种形式的正则表达式：

```
s/^\s+//;
s/\s+$//;
```

因为它几乎总是最快的，而且显然最容易理解。

HTML 相关范例

HTML-Related Examples

在第 2 章，我们曾讨论过把纯文本转换为 HTML 的例子（☞67），其中要使用正则表达式从文本中提取 E-mail 地址和 http URL。本节来看一些与 HTML 相关的其他处理。

匹配 HTML Tag

Matching an HTML Tag

最常见的办法就是用「<[^>]+>」来匹配 HTML 标签。它通常都能工作，例如下面这段用来去除标签的 Perl 语句：

```
$html = ~ s/<[^>]+>//g;
```

如果 tag 中含有‘>’，它就不能正常匹配了，而这样的 tag 明明是合乎 HTML 规范的：<input name=dir value=">">。虽然这种情况很少见，也不为大家推荐，但 HTML 语言确实容许在引号内的 tag 属性中出现非转义的‘<’和‘>’。这样，简单的「<[^>]+>」就无法匹配了，得想个聪明点的办法。

‘<…>’中能够出现引用文本和非引用形式的“其他文本（other stuff）”，其中包括除了‘>’和引号之外的任意字符。HTML 的引文可以用单引号，也可以用双引号。但不容许转义嵌套的引号，所以我们可以直接用「"[^"]*"」和「'[^']*'」来匹配。

把这些和“其他文本”表达式「[^'">]」合起来，我们得到：

「<("[^"]*"|'[^']*'|[^'">])*>」

这个表达式可能有点难看懂，那么加上注释，按宽松排列格式来看：

```
<                         # 开始尖括号 "<"
    (                     #    任意数量的 ...
      "[^"]*"             #        双引号字符串
      |                   #        或者是...
      '[^']*'             #        单引号字符串
      |                   #        或者是...
      [^'">]              #        "其他文本"
    )*                    #
 >                        # 结束尖括号 ">"
```

这个表达式相当漂亮，它会把每个引用部分单作为一个单元，而且清楚地说明了在匹配的什么位置容许出现什么字符。这个表达式的各部分不会匹配重复的字符，因此不存在模糊性，也就不需要担心发生前面例子中出现的，“不小心冒出来（sneaking in）”非期望匹配。

不知你是否注意到了，最开始的两个多选分支的引号中使用了「*」，而不是「+」。引用字符串可能为空（例如‘alt=""’），所以要用「*」来处理这种情况。但不要在第三个多选分支中用「*」取代「+」，因为「[^'">]」只接受括号外的「*」的限定。给它添加一个加号得到「([^'">]+)*」，可能导致非常奇怪的结果，我不期望读者现在就能理解，下一章（☞226）会详细讲解它。

在使用 NFA 引擎时，我们还需要考虑关于效率的问题：既然没有用到括号匹配的文本，我们可以把它们改为非捕获型括号（☞137）。因为多选分支之间不存在重复，如果最后的「>」无法匹配，那么回头来尝试其他的多选分支也是徒劳的。如果一个多选分支能够在某个位置匹配，那么其他多选分支肯定无法在这里匹配。所以，不保存状态也无所谓，这样做还可以更快地导致失败，如果找不到匹配结果的话。我们可以用固化分组「(?>…)」而不是非捕获型括号（或者用占有优先的星号限定）。

匹配 HTML Link

Matching an HTML Link

假设我们需要从一份文档中提取 URL 和链接文本，例如下面的文本中标记的内容：

```
…<a href="http://www.oreilly.com">O'Reilly Media</a>…
```

因为<A> tag 的内容可能相当复杂，我会分两步实现这个任务。第一个是提取<A> tag 内部的内容，也就是链接文本，然后从<A> tag 中提取 URL 地址。

实现第一步有个简单办法，就是在点号通配模式下应用不区分大小写的「<a\b([^>]+)>(.*?)」，这里使用了忽略优先量词。它会把<A>的内容放入$1，把链接文本放入$2。当然，像之前一样，我不应该用「[^>]+」，而应该使用前几节中的表达式。不过在本节，我会继续使用这个简单的形式，因为这样正则表达式更短，也更容易讲解。

<A>的内容存入字符串之后，就可以用独立的正则表达式来检查它们。其中，URL 是 href=*value* 属性的值。之前已经说过，HTML 容许等号的任意一侧出现空白字符，值可以

以引用形式出现，也可以以非引用形式出现。下面的 Perl 代码用来输出变量$Html 中的链接。

```
# 请注意：while(...)中的正则表达式是简化的形式，请参见正文
while ($Html =~ m{a\b([^>]+)>(.*?)</a>}ig)
{
  my $Guts = $1; # 把匹配结果存入 ...
  my $Link = $2; #  ....对应变量
  if ($Guts =~ m{
                \b HREF          #  "href" 属性
                \s* = \s*        #  "=" 两端可能出现空白字符
                (?:              #  其值为...
                  "([^"]*)"      #      双引号字符串
                  |              #      或者是...
                  '([^']*)'      #      单引号字符串
                  |              #      或者是...
                  ([^'">\s]+)    #      "其他文本"
                )                #
               }xi)
  {
     my $Url = $+;     #  获得$1、$2 等中实际参与匹配的编号最大的捕获型括号的内容
     print "$Url with link text: $Link\n";
  }
}
```

有几点需要注意：

- 我们为匹配值的每个多选结构都添加了括号，来捕获确切的文本。
- 因为我使用了某些括号来捕获文本，在不需要捕获的地方我使用非捕获型括号，这样做既清楚又高效。
- “其他字符”部分排除了空白字符，也排除了引号和‘>’。
- 因为需要捕获整个 href 的值，这里使用了「+」来限制“其他文本”多选分支。这是否会和第 200 页对其他字符应用「+」一样导致“非常奇怪的结果”呢？不会，因为这外面没有直接作用于整个多选结构的量词。其中的细节同样会在下一章讨论。

根据具体文本的不同，最后，URL 可能保存在$1、$2 或者$3 中。此时其他捕获型括号就为空或是未定义。Perl 提供了特殊变量$+，代表$1、$2 之类中编号最靠后的捕获文本。在本例中，这就是我们真正需要的 URL。

Perl 中的$+很方便，其他语言也提供了其他办法来选择捕获的 URL。常用的程序语言结构就可以检查捕获型括号，找到需要的内容。如果能够支持，命名捕获（☞138）最适用于干这个，就像 204 页的 VB.NET 的例子那样（幸亏.NET 提供了命名捕获，因为它的$+有问题，☞424）。

检查 HTTP URL

Examining an HTTP URL

现在我们得到了 URL 地址，来看看它是否是 HTTP URL，如果是，就把它分解为主机名（hostname）和路径（path）两部分。因为已经有了 URL，任务就比从随机文本中识别 URL 要简单许多，识别的程序要难许多，这将在后文介绍。

所以，如果拿到一个 URL，我们需要能够将它拆分为各个部分。主机名是「^http://」之后和第一个反斜线（如果有的话）之前的内容，而路径就是除此之外的内容：「^http://([^/]+)(/.*)?$」。

URL 中有可能包含端口号，它位于主机名和路径之间，以一个冒号开头：「^http://([^/:]+)**(:(\d+))?**(/.*)?$」。

下面是一个分解 URL 的 Perl 代码：

```
if ($url =~ m{^http://([^/:]+)(:(\d+))?(/.*)?$}i)
{
  my $host = $1;
  my $port = $3 || 80;    # 如果存在，就使用$3；否则默认为 80
  my $path = $4 || "/";   # 如果存在，就使用$4；否则默认为"/"
  print "Host: $host\n";
  print "Port: $port\n";
  print "Path: $path\n";
} else {
  print "Not an HTTP URL\n";
}
```

验证主机名

Validating a Hostname

在上面的例子中，我们用「[^/:]+」来匹配主机名。不过，在第 2 章中（☞76）我们使用的是更复杂的「[-a-z]+(\.[-a-z]+)*\.(com|edu|…|info)」。做同样的事情，复杂程度为什么会有这么大的差别？

而且，虽然二者都用来“匹配主机名”，方法却大不相同。从已知文本（例如，从现成的 URL 中）中提取一些信息是一回事，从随机文本中准确提取同样信息是另一回事。

而且，在上例中我们假设，‘http://’之后就是主机名，所以用「[^/:]+」来匹配就是理所当然的。但是在第 2 章的例子中，我们使用正则表达式从随机文本中寻找主机名，所以它必须更加复杂。

现在从另外一个角度来看主机名的匹配，我们可以用正则表达式来验证主机名。也就是说，我们需要知道，一串字符是否是形式规范、语意正确的主机名。按规定，主机名由点号分

VB.NET 中的 link 检查程序

下面的程序会列出 *Html* 变量中的链接：

```
Imports System.Text.RegularExpressions

' 设置循环中将会遇到的正则表达式
Dim A_RRegex as Regex = New Regex(                              _
             "<a\b(?<guts>[^>]+)>(?<Link>.*?)</a>",  _
             RegexOptions.IgnoreCase)
Dim GutsRegex as Regex = New Regex( _
   "\b HREF                    (?#   'href' 属性          ) " & _
   "\s* = \s*                  (?#   '=' 可能存在空白字符 ) " & _
   "(?:                        (?#   其值为...            ) " & _
   "  ""(?<url>[^""]*)""       (?#   双引号字符串         ) " & _
   "  |                        (?#   或者是...            ) " & _
   "  '(?<url>[^']*)'          (?#   单引号字符串         ) " & _
   "  |                        (?#   或者是...            ) " & _
   "  (?<url>[^'"">\s]+)       (?#   '其他文本'           ) " & _
   ")                          (?#                        ) " , _
   RegexOptions.IgnoreCase Or RegexOptions.IgnorePatternWhitespace)

' 现在检查 'Html' 变量 ...
Dim CheckA as Match = A_Regex.Match(Html)

' For each match within ...
While CheckA.Success
   ' 已匹配 <a> tag，现在检查 URL
   Dim UrlCheck as Match = _
      GutsRegex.Match(CheckA.Groups("guts").Value)
   If UrlCheck.Success
      ' 已经匹配完毕，得到 URL/link
      Console.WriteLine("Url " & UrlCheck.Groups("url").Value & _
                        " WITH LINK " & CheckA.Groups("Link").Value)
   End If
   CheckA = CheckA.NextMatch
End While
```

需要注意的几点：

- 在 VB.NET 中使用正则表达式，需要首先执行对应的 `Imports` 语句，告诉编译器应当导入的库文件。
- 程序中使用了「(?#…)」风格的注释，因为 VB.NET 中加入换行符很不方便，所以普通的‘#’注释会延伸到下一个换行符或者字符串的结尾（第一种情况即意味着正则表达式剩下的所有内容都作为注释）。为了使用正常的「#…」注释，请在每一行的结尾添加`&chr(10)`（☞420）。
- 表达式中的每个双引号都需要以‘""’表示（☞103）。
- 两个表达式都用到了命名捕获，`Groups("url")`比 `Groups(1)`和 `Groups(2)`之类更为清晰。

隔的部分组成，每个部分可以包括 ASCII 字符、数字和连字符，但是不能以连字符作为开头和结尾。所以，我们可以在不区分大小写的模式下使用这个正则表达式：「[a-z0-9]|[a-z0-9][-a-z0-9]*[a-z0-9]」。结尾的后缀部分（'com'、'edu'、'uk' 等）只有有限多个可能，这在第 2 章的例子中提到过。结合起来，下面的正则表达式就能够匹配一个语意正确的主机名：

```
^
 (?i)          #  进行不区分大小写的匹配
  #  零个或多个据点分隔的部分
 (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]*[a-z0-9]\. )+
  #  然后是结尾的后缀部分...
 (?: com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z] )
$
```

因为存在长度的限制，能够由这个正则表达式匹配的可能并不是合法的主机名：每个部分不能超过 63 个字符。也就是说，「[-a-z0-9]*」应该改为「[-a-z0-9]{0,61}」。

还需要做最后的改动。按规定，只包括后缀的主机名同样是语意正确的。但实践证明，这些"主机名"不存在，但是对于两个字母的后缀来说情况可不是如此。例如，安哥拉的域名 'ai' 就有一个 Web 服务器 *http://ai/*。我见过其他这样的链接：cc、co、dk、mm、ph、tj、tv 和 tw。

如果希望匹配这些特殊情况，应该把中间的「(?:…)+」改为「(?:…)*」：

```
^
 (?i)          #  进行不区分大小写的匹配
  #  零个或多个据点分隔的部分
 (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]{0,61}[a-z0-9]\. )*
  #  然后是结尾的后缀部分...
 (?: com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z] )
$
```

现在它可以用来验证包含主机名的字符串了。因为这是我们想出的与主机名相关的三个正则表达式中最复杂的，你也许会想，不要这些锚点，可能比之前那个从随机文本中提取主机名的表达式更好。但情况并非如此。这个正则表达式能匹配任意双字母单词，正因为如此，第 2 章中不那么精妙的正则表达式的实际效果更好。但是在下一节我们会看到，某些情况下它仍然不够完善。

在真实世界中提取URL

Plucking Out a URL in the Real World

供职于Yahoo! Finance时，我曾写过处理收录的财经新闻和数据的程序。新闻通常是以纯文本格式提供的，我的程序将其转化为HTML格式以便于显示（如果你在过去10年中曾经在*http://finance.yahoo.com*浏览过财经新闻，没准看过我处理过的新闻）。

因为接受的数据的“格式”（其实是无格式）很杂乱，从纯文本中识别（recognize）出hostname和URL又比验证（validate）它们困难得多，这任务就很不轻松。前面的内容并没有体现这一点，在本节，你会看到我在Yahoo!用来解决这个问题的程序。

这个程序从文本中提取几种类型的URL——`mailto`、`http`、`https`和`ftp`。如果我们在文本中找到‘`http://`’，就知道这肯定是一个URL的开头，所以我们可以直接用「`http://[-\w]+(\.\w[-\w]*)+`」来匹配主机名。我们知道，要处理的文本肯定是ASCII编码的英文字母，所以完全可以用「`-\w`」来取代「`-a-z0-9`」。「`\w`」同样可以匹配下画线，在某些系统中，它还可以匹配所有的Unicode字符，但是我们知道，这个程序在运行时不会遇到这些问题。

不过，URL通常不是以`http://`或者`mailto:`开头的，例如：

```
…visit us at www.oreilly.com or mail to orders@oreilly.com…
```

在这种情况下，我们需要加倍小心。我在Yahoo!使用的正则表达式与前面那节的非常相似，只是有一点点不同：

```
(?i: [a-z0-9] (?:[-a-z0-9]*[a-z0-9])? \. )+    # 子域名s
                         #  .com之类的后缀，要求小写
(?-i: com\b
    | edu\b
    | biz\b
    | org\b
    | gov\b
    | in(?:t|fo)\b     #  .int 或者.info
    | mil\b
    | net\b
    | name\b
    | museum\b
    | coop\b
    | aero\b
    | [a-z][a-z]\b     #  双字母国家代码
)
```

在这个正则表达式中，我们用「`(?i:…)`」和「`(?-i:…)`」用来规定正则表达式的某个部分是否区分大小写（☞135）。我们希望匹配‘`www.OReilly.com`’，但不是‘`NT.TO`’这样的股票

代码（NT.TO 是北电网络在多伦多证券交易市场的代号，因为要处理的是财经新闻和数据，这样的股票代码很多）。按规定，URL 的结尾部分（例如‘.com’）可能是大写的，但我不准备处理这些情况。因为我需要保持平衡——匹配期望的文本（尽可能多的 URL），忽略不期望的文本（股票代码）。我希望「(?-i:...)」只包括国家代码，但是在现实中，我们没有遇到大写的 URL 地址，所以不必这么做。

下面是从纯文本中查找 URL 的框架，我们可以在其中添加匹配主机名的子表达式：

```
\b
     #  匹配开头部分  (proto://hostname，或直接是 hostname)
 (
     #  ftp://、http:// 或 https:// 开头部分
     (ftp|https?)://[-\w]+(\.\w[-\w]*)+
   |
     #  或者是用更准确的子表达式找到 hostname
     full-hostname-regex
)

     #  可能出现端口号
 ( : \d+ )?

     #  下面部分可能出现，以/开头
 (
    / path-part
)?
```

我还没有谈论过正则表达式的 path（路径）部分，它接在主机名后面（例如 http://www.orielly.com/catalog/regex/中的划线部分）。path 是最难正确匹配的文本，因为它需要一些猜测才能做得很漂亮。我们在第 2 章说过，通常出现在 URL 之后的文本也能被作为 URL 的一部分。例如：

```
Read his comments at http://www.oreilly.com/ask_tim/index.html. He...
```

我们观察之后就会发现，在‘index.html’之后的句号是一个标点，不应该作为 URL 的一部分，但是‘index.html’中的点号却是 URL 的一部分。

肉眼很容易分辨这两种情况，但程序做起来却很难，所以必须想些聪明的办法来尽可能好地解决问题。第 2 章的例子使用逆序环视来确保 URL 不会以句末的句号结尾。我在 Yahoo! Finance 写程序时还没有逆序环视，所以我用的办法要复杂的多，不过效果是一样的。代码在下一页。

示例 5-1：从财经新闻中提取 URL

```
\b
# 匹配开头部分 (proto://hostname，或直接是 hostname)
 (
   # ftp://、http://或 https:// 开头部分
   (ftp|https?)://[-\w]+(\.\w[-\w]*)+
  |
   # 或者是用更准确的子表达式找到hostname
   (?i: [a-z0-9] (?:[-a-z0-9]*[a-z0-9])? \. )+        # sub domains
   # .com 之类的后缀．要求小写
   (?-i: com\b
       | edu\b
       | biz\b
       | gov\b
       | in(?:t|fo)\b     # .int 或者.info
       | mil\b
       | net\b
       | org\b
       | [a-z][a-z]\b     # 双字母国家代码
    )
)
# 可能出现端口号
 ( : \d+ )?

# 剩下的部分可能出现，以/开头 ...
 (
     /
    # 虽然很复杂，但确实管用
    [^.!,?;"'<>()\[\]{}\s\x7F-\xFF]*
    (?:
        [.!,?]+ [^.!,?;"'<>()\[\]{}\s\x7F-\xFF]+
     )*
)?
```

这里用到的办法与第 2 章第 75 页用到的办法有很多不同，比较起来也很有意思。下一页里使用此表达式的 Java 程序详细介绍了它的构造。

在实际生活中，我怀疑自己是否会写这样繁杂的正则表达式，但是作为取代，我会建立一个正则表达式“库（library）”，需要时取用。这方面一个简单的例子就是第 76 页的 `$HostnameRegex`，以及下面的补充内容。

扩展的例子

Extended Examples

下面的几个例子讲解了一些关于正则表达式的重要诀窍。讨论会稍微多一些，关于解决办法和错误思路的着墨也会更多一些，最终会给出正确答案。

在 Java 中通过变量构建正则表达式

```
String SubDomain  = "(?i:[a-z0-9]|[a-z0-9][-a-z0-9]*[a-z0-9])";
String TopDomains = "(?x-i:com\\b             \n" +
                    "     |edu\\b             \n" +
                    "     |biz\\b             \n" +
                    "     |in(?:t|fo)\\b      \n" +
                    "     |mil\\b             \n" +
                    "     |net\\b             \n" +
                    "     |org\\b             \n" +
                    "     |[a-z][a-z]\\b      \n" + // country codes
                    ")                        \n";
String Hostname = "(?:" + SubDomain + "\\.)+" + TopDomains;

String NOT_IN   = ";\"'<>()\\[\\]{}\\s\\x7F-\\xFF";
String NOT_END  = "!.,?";
String ANYWHERE = "[^" + NOT_IN + NOT_END + "]";
String EMBEDDED = "[" + NOT_END + "]";
String UrlPath =  "/"+ANYWHERE + "*("+EMBEDDED+"+"+ANYWHERE+"+)*";
String Url =
  "(?x:                                                   \n"+
  "  \\b                                                  \n"+
  "  ## 匹配 hostname                                      \n"+
  "  (                                                    \n"+
  "    (?: ftp | http s? ): // [-\\w]+(\\.\\w[-\\w]*)+ \n"+
  "   |                                                   \n"+
  "    " + Hostname + "                                   \n"+
  "  )                                                    \n"+
  "    # 可能出现端口号                                    \n"+
  "  (?: :\\d+ )?                                         \n"+
  "                                                       \n"+
  "    # 下面的部分可能出现，以\开头                        \n"+
  "  (?: " + UrlPath + ")?                                \n"+
  " )";
// 现在把正则表达式编译为正则对象
Pattern UrlRegex = Pattern.compile(Url);
// 现在准备在文本中应用，寻找url...
  ......
```

保持数据的协调性

Keeping in Sync with Your Data

我们来看一个长一点的例子，它有点极端，但很清楚地说明了保持协调的重要性（同时提供了一些保持协调的方法）。

假设，需要处理的数据是一系列连续的 5 位数美国邮政编码（ZIP Codes），而需要提取的是以 44 开头的那些编码。下面是一点抽样，我们需要提取的数值用粗体表示：

03824531449411615213**44182**95053**44272**752010217443235

最容易想到的「\d\d\d\d\d」，它能匹配所有的邮政编码。在 Perl 中可以用**@zips=m/\d\d\d\d\d/g;**来生成以邮政编码为元素的 list（为了让这些例子看起来更整洁，我们假设需要处理的文本在 Perl 的默认目标变量$_中，见☞79）。如果使用其他语言，也只需要循环调用正则表达式的 find 方法。我们关注的是正则表达式本身，而不是语言的实现机制，所以下面继续使用 Perl。

回到「\d\d\d\d\d」，下面提到的这一点很快就会体现出其价值：在整个解析过程中，这个正则表达式任何时候都能够匹配——绝对没有传动装置的驱动和重试（我假设所有的数据都是规范的，此假设与具体情况密切相关）。

很明显，把「\d\d\d\d\d」改为「44\d\d\d」来查找以 44 开头的邮政编码不是个好办法——匹配失败之后，传动装置会驱动前进一个字符，对「44…」的匹配不再是从每个邮政编码的第一位开始。「44\d\d\d」会错误地匹配‘…531**44941**16…’。

当然，我们可以在正则表达式的开头添加「\A」，但是这样只能对付一行文本中的第一个邮政编码。我们需要手动保持正则引擎的协调，才能忽略不需要的邮政编码。这里的关键是，要跳过完整的邮政编码，而不是使用传动装置的驱动过程（bump-along）来进行单个字符的移动。

根据期望保持匹配的协调性

下面列举了几种办法用来跳过不需要的邮政编码。把它们添加到正则表达式「44\d\d\d」之前，可以获得期望的结果。非捕获型括号用来匹配不期望的邮政编码，这样能够快速地略过它们，找到匹配的邮政编码，在第一个$1 的捕获括号中：

「**(?:**[^4]\d\d\d\d**|**\d[^4]\d\d\d**)***…」

　　这种硬办法（brute-force method）主动略过非 44 开头的邮政编码（当然，用「[1235-9]」替代「[^4]」可能更合适，但我之前说过，假设处理的是规范的数据）。注意，我们不能使用「**(?:**[^4][^4]\d\d\d**)***」，因为它不会匹配（也就无法略过）43210 这样不期望的邮政编码。

「**(?:**(?!44)\d\d\d\d\d**)***…」

　　这个办法跳过非 44 开头的邮政编码。其中的想法与之前并无差别，但用正则表达式写出来就显得大不一样。比较这两段描述和相关的正则表达式就会发现，在这里，期望的邮政编码（以 44 开头）导致逆序环视（?!44）失败，于是略过停止。

「(?:\d\d\d\d\d)*?…」

这个办法使用忽略优先量词，只有在需要的时候才略过某些文本。我们把它放在真正需要匹配的正则表达式前面，所以如果那个表达式失败，它就会匹配一个邮政编码。忽略优先「(…)*?」导致这一切的发生。因为存在忽略优先量词，「(?:\d\d\d\d\d)」甚至都不会尝试匹配，在后面的表达式失败之前。星号确保了，它会重复失败，直到最终找到匹配文本，这样就能只跳过我们希望跳过的文本。

把这个表达式和「(44\d\d\d)」合起来，就得到：

```
@zips=m/(?:\d\d\d\d\d)*?(44\d\d\d)/g;
```

它能够提取以 44 开头的邮编，而主动跳过其他的邮编（在“**@array = m/…/g**”的情况下，Perl 会用每次尝试中找到的匹配文本来填充这个数组，☞311）。这个表达式能够重复应用于字符串，因为我们知道每次匹配的“起始匹配位置”都是某个邮政编码的开头位置，也就保证下一次匹配是从一个邮政编码的开始，这正是正则表达式期望的。

不匹配时也应当保证协调性

我们是否能保证，每次正则表达式都在邮政编码字符串的开头位置应用？显然不是！我们手动跳过了不符合要求的邮政编码，可一旦不需要继续匹配，本轮匹配失败之后自然就是驱动过程和重试，这样就会从邮政编码字符串之中的某个位置开始——我们的方法不能处理这种情况。

再来看数据样本：

03824531449411615213**44182**95035**44272**7520102174**4323**5

匹配的代码以粗体标注（第三组不符合要求），主动跳过的代码以下画线标注，通过驱动过程-重试略过的字符也标记出来。在 44272 匹配之后，目标文本中再也找不到匹配，所以本轮尝试宣告失败。但总的尝试并没有宣告失败。传动机构会进行驱动，从字符串的下一个字符开始应用正则表达式，这样就破坏了协调性。在第四次驱动之后，正则表达式略过 10217，错误地匹配 44323。

如果在字符串的开头应用，这三个表达式都没有问题，但是传动装置的驱动过程会破坏协调性。如果我们能取消驱动过程，或者保证驱动过程不会添麻烦，问题就解决了。

办法之一是禁止驱动过程，即在前两种办法中的「(44\d\d\d)」之后添加「?」，将其改为匹配优先的可选项。这样，刻意安排的「(?:(?!44)\d\d\d\d\d)*…」或「(?:[^4]\d\d\d\d|\d

「[^4]\d\d\d)*…」就只会在两种情况下停止：发生符合要求的匹配，或者邮政编码字符串结束（这也是此方法不适用于第三个表达式的原因）。这样，如果存在符合要求的邮政编码，「(44\d\d\d)?」就能匹配，而不会强迫回溯。

这个办法仍然不够完善。原因之一是，即便目标字符串中没有符合要求的邮政编码，也会匹配成功，接下来的处理程序会变得更复杂。不过，其优点在于速度很快，因为不需要回溯，也不需要传动装置进行任何驱动过程。

使用\G 保证协调

更通用的办法是在这三个表达式末尾添加「\G」(☞130)。因为每个表达式的每次匹配都以符合要求的邮政编码结尾，下次匹配开始时就不会进行驱动。而如果有驱动过程，开头的「\G」会立刻导致匹配失败，因为在大多数流派中，只有在未发生驱动过程的情况下，它才能成功匹配（但在 Ruby 和其他规定「\G」表示“本次匹配起始位置”的流派中不成立☞131）

所以第二个表达式就变成了：

```
@zips = m/\G(?:(?!44)\d\d\d\d\d)*(44\d\d\d)/g;
```

匹配之后不需要进行任何特殊检查。

本例的意义

我首先承认，这个例子有点极端，不过，它包含了许多保证正则表达式与数据协调性的知识。如果现实生活中需要处理这样的问题，我可能不会完全用正则表达式来解决。我会直接用「\d\d\d\d\d」来提出每个邮政编码，然后检查它是否以‘44’开头。在 Perl 中是这样：

```
@zips = ( );      #  确保数组为空

while (m/(\d\d\d\d\d)/g) {
   $zip = $1;
   if (substr($zip, 0, 2) eq "44") {
      push @zips, $zip;
   }
}
```

对「\G」有兴趣的读者请参考 132 页的补充内容，尽管本书写作时只能举 Perl 的例子。

解析 CSV 文件

Parsing CSV Files

解析 CSV（逗号分隔值）文件有点麻烦，因为每个程序都有自己的 CSV 文件格式。首先来看如何解析 Microsoft Excel 生成的 CSV 文件，然后再看其他格式(注 3)。幸运的是，Microsoft 的格式是最简单的。以逗号分隔的值要么是“纯粹的”（仅仅包含在括号之前），要么是在双引号之间（这时数据中的双引号以一对双引号表示）。

下面是个例子：

```
Ten Thousand,10000, 2710 ,,"10,000","It's ""10 Grand"", baby",10K
```

这一行包含七个字段（fields）：

```
Ten·Thousand
10000
·2710·
空字段
10,000
It's·"10·Grand",·baby
10K
```

为了从此行解析出各个字段，我们的正则表达式需要能够处理两种格式。非引号格式包含引号和逗号之外的任何字符，可以用「`[^",]+`」匹配。

双引号字段可以包含逗号、空格，以及双引号之外的任何字符。还可以包含连在一起的两个双引号。所以，双引号字段可以由「`"…"`」之间任意数量的「`[^"]|""`」匹配，也就是「`"(?:[^"]|"")*"`」（为效率考虑，我们可以使用固化分组「`(?>…)`」来替代「`(?:…)`」，不过这个话题留到下一章☞259）。

综合起来，「`[^",]+|"(?:[^"]|"")*"`」能够匹配一个字段。这可能有点难看懂，下面我们给出宽松排列（☞111）格式：

```
#  引号和逗号之外的文本...
[^",]+
#  ...或者是...
|
#  ...双引号字段（其中容许出现连在一起的成对双引号）
" #  起始双引号
  (?: [^"] | "" )*
" #  结束双引号
```

注 3：在第 6 章，讨论完效率问题之后，会给出处理 Microsoft Excel 的最终程序（☞271）。

现在这个表达式可以实际应用到包含 CSV 文本行的字符串上了，但如果我们希望真正利用匹配结果，就应该知道具体是哪个多选分支匹配了。如果是双引号字符串，就需要去掉首尾两端的双引号，把其中紧挨着的两个双引号替换为单个双引号。

我能想到的办法有两个。其一是检查匹配结果的第一个字符是否双引号，如果是，则去掉第一个和最后一个字符（双引号），然后把中间的‘""’替换为‘"’。这办法够简单，但如果使用捕获型括号会更简单。如果我们给捕获字段的每个子表达式添加捕获型括号，可以在匹配之后检查各个分组的值：

```
#  引号和逗号之外的文本...
( [^",]+ )
#  ... 或者是...
   |
#  ...双引号字段（其中容许出现连在一起的成对双引号）
"  #  起始双引号
 ( (?: [^"] | "" )* )
"  #  结束双引号
```

如果是第一个分组捕获，则不需要进行任何处理，如果是第二个分组，则只需要把‘""’替换为‘"’即可。

下面给出 Perl 的程序，稍后（找出某些 bug 之后）给出 Java 和 VB.NET（在第 10 章给出 PHP 的程序☞480）。下面是 Perl 程序，假设数据位于`$line`中，而且已经去掉了结尾的换行符（换行符不属于最后的字段！）：

```
while ($line =~ m{
           #  引号和逗号之外的文本...
           ( [^",]+ )
           #  ...或者是...
             |
           #  ...双引号字段（其中容许出现连在一起的成对双引号）
           "  #  起始双引号
             (     (?: [^"] | "" )* )
           " #  结束双引号
        }gx)
{
  if (defined $1) {
      $field = $1;
  } else {
      $field = $2;
      $field =~ s/""/"/g;
  }
  print "[$field]";   #  输出$field供调试
  现在可以处理$field了...
}
```

将其应用于测试数据，结果为：

```
[Ten·Thousand][10000][·2710·][10,000][It's·"10·Grand",·baby][10K]
```

看来没问题，但不幸的是它不会输出为空的第四个字段。如果“处理$field”是将字段的值存入数组，完成后访问数组的第五个元素得到第五个字段（“10,000”）。这显然不对，因为数组的元素与空字段不对应。

想到的第一个办法是把「[^",]+」改为「[^",]*」，这看来是显而易见的，但它正确吗？

测试一下，下面是结果：

```
[Ten·Thousand][][10000][][·2710·][][][][10,000][][][It's·"10·Grand", …
```

哇，现在出来了一堆空字段！仔细检查检查，就不会这么吃惊。「(…)*」的匹配可以不占用任何字符。如果真的遇到空字段，确实能匹配，那么考虑第一个字段匹配之后的情况呢，此时正则表达式从‘Ten·Thousand▲,10000…’开始应用。如果表达式中没有元素可以匹配逗号（就本例来说），就会发生**长度为0**的成功匹配。实际上，这样的匹配可能有无穷多次，因为正则引擎可能在同一位置重复这样的匹配，现代的正则引擎会强迫进行驱动过程，所以同一位置不会发生两次长度为 0 的匹配（☞131）。所以每个有效匹配之间还有一个空匹配，在每个引号字段之前会多出一个空匹配（而且数组末尾还会有一个空匹配，只是此处没有列出来）。

分解驱动过程

要解决问题，我们就不能依赖传动机构的驱动过程来越过逗号。所以，我们需要手工来控制。能想到的办法有两个：

1. 手工匹配逗号。如果采取此办法，需要把逗号作为普通字段匹配的一部分，在字符串中“迈步（pace ourselves）”。

2. 确保每次匹配都从字段能够开始的位置开始。字段可以从行首，或者是逗号开始。

可能更好的办法是把两者结合起来。从第一种办法（匹配逗号本身）出发，只需要保证逗号出现在第一个字段之外的所有字段开头。或者，保证逗号出现在最后一个字段之外的所有字段的末尾。可以在表达式前面添加「^|,」，或者后面添加「$|,」，用括号控制范围。

在前面添加，就得到：

```
(?:^|,)
(?:
    #  引号和逗号之外的文本....
    ( [^",]* )
  #  ...或者是..
  |
    #  ... 双引号字段（其中容许出现连在一起的成对双引号）
    "  #  起始双引号
     ( (?: [^"] | "" )* )
    "  #  结束双引号
)
```

看起来它应当没错，但实际的结果却是：

```
[Ten·Thousand][10000][·2710·][][][000][][·baby][10K]
```

而我们期望的是：

```
[Ten·Thousand][10000][·2710·][][10,000][It's·"10·Grand",·baby][10K]
```

问题出在哪里呢？似乎是双引号字段没有正确处理，所以问题出在它身上，对吗？不对，问题在前面。或许 176 页的告诫有所帮助：**如果多个多选分支能够在同一位置匹配，必须小心地排列顺序。**第一个多选分支「[^",]*」不需要匹配任何字符就能成功，除非之后的元素强迫，否则第二个多选分支不会获得尝试的机会。而这两个多选分支之后没有任何元素，所以第二个多选分支永远不会得到尝试的机会，这就是问题所在！

哇，现在我们已经找到了问题所在。OK，交换一下多选分支的顺序：

```
(?:^|,)
(?: #  或者是匹配双引号字段（其中容许出现连在一起的成对双引号）...
       " #  (起始双引号)
        ( (?: [^"] | "" )* )
       " #  (起始双引号)
  |
    #  ... 或者是引号和逗号之外的文本...
         ( [^",]* )
)
```

对了！至少对测试数据来说是对了。如果数据变了，还是这样吗？本节的标题是“分解驱动过程”，而最保险的办法就是以完整测试作为基础的思考，故可以用「\G」来确保每次匹配从上一次匹配结束的位置开始。考虑到构建和应用正则表达式的过程，这样做应该绝对没问题。如果在表达式开始添加「\G」，就会禁止引擎的驱动过程。我们希望这样修改不会出问

题，但是结果并非如此。之前输出

```
[Ten·Thousand][10000][·2710·][][][000][][·baby][10K]
```

的正则表达式添加\G之后，得到

```
[Ten·Thousand][10000][·2710·][][]
```

如果起初没看明白，这样看会更明显。

CSV Processing in Java

这里有一个使用Sun的java.util.regex解析CSV的例子。这段程序着眼于简洁的、更有效的版本——第8章（☞401）将会介绍。

```
import java.util.regex.*;
        .
        .
        .
String regex = // 把双引号字段存入group(1)、非引号字段存入group(2)
   "\\G(?:^|,)                                       \n"+
   "(?:                                              \n"+
   "   # 要么是双引号字段...                          \n"+
   "   \"            # 字段起始双引号                  \n"+
   "    (  (?: [^\"]++ | \"\" )*+ )                   \n"+
   "   \"            # 字段结束双引号                  \n"+
   " |# ... 要么是 ...                               \n"+
   "           # 非引号非逗号文本 ...                  \n"+
   "    ( [^\",]*)                                   \n"+
   " )                                               \n";
// 创建使用上面正则表达式的matcher，暂时不指定需要应用的文本
Matcher mMain = Pattern.compile( regex, Pattern.COMMENTS).matcher("");

   // 为「""」创建一个matcher，暂时不指定需要应用的文本
Matcher mQuote = Pattern.compile("\"\"").matcher("");
        .
        .
        .
// 上面都是准备工作，下面的代码逐行处理文本
mMain.reset( line); //下面处理line中的CSV文本
while ( mMain.find())
{
    String field;
    if ( mMain.start(2) >= 0)
         field = mMain.group(2);     // 非引号字段，直接使用
    else
         // 引号字段，替换其中的成对双引号
         field = mQuote.reset(mMain.group(1)).replaceAll("\"");
    // 处理字段...
    System.out.println("Field [" + field + "]");
}
```

另一个办法

本节的开头提到有两种办法正确匹配各个字段。之二是确保匹配只能在容许出现字段的地方开始。从表面上看，这类似于添加「^|,」，只是使用了逆序环视「(?<=^|,)」。

不幸的是，按照第 3 章（☞133）的解释，即使可以使用逆序环视，也不见得能够使用变长的逆序环视，所以此方法可能无法使用。如果问题在于长度可变，我们可以把「(?<=^|,)」替换为「(?:^|(?<=,))」，但是相比第一种办法，它太麻烦了。而且，它仍然依赖传动装置的驱动过程来越过逗号，如果别的地方出了什么差错，它会容许在'…"10,▲000"…'处的匹配。总的来说就是，不如第一种办法保险。

不过我们可以略施小计——要求匹配在逗号之前（或者是一行结束之前）**结束**。在表达式结尾添加「(?=$|,)」可以确保它不会进行错误的匹配。实际生活中，我会这样做吗？直率地说我觉得第一种方法很合用，所以遇到这种情况我可能不会采取第二种办法，不过如果需要，这技巧却是很有用的。

进一步提高效率

尽管在下一章之前都不会谈论效率，但对于支持固化分组（☞139）的系统，我还是愿意在这里给出提高效率的修改：把匹配双引号字段的子表达式从「(?:[^"]|"")*」改为「(?>[^"]+|"")*」。下一页用 VB.NET 的例子做了说明。

如果像 Sun 的 Java regex package 那样支持占有优先量词（☞142），也可以使用占有优先量词。Java CSV 程序的补充内容说明了这一点。

这些修改背后的道理会在下一章讲解，最终我们会在 271 页给出效率最高的办法。

其他 CSV 格式

Microsoft 的 CSV 格式很流行，因为它是 Microsoft 的 CSV 格式，但其他程序可能有不同格式，我见过的情况还有：

- 使用任意字符，例如';'或者制表符作为分隔。（不过这样名字还能叫"逗号分隔值"吗？）
- 容许分隔符之后出现空格，但不把它们作为值的一部分。

- 用反斜线转义引号（例如用‘\"’而不是‘""’类表示值内部的引号）。通常这意味着反斜线可以在任何字符前出现（并忽略）。

这些变化都很容易处理。第一种情况只需要把逗号替换为对应的分隔符，第二种只需要在第一个分隔符之后添加「\s*」，例如以「(?:^|,\s*)」开头。

第三种情况，我们可以用之前的办法（☞198），把「[^"]+|""」替换为「[^\\"]+|\\.」。当然，我们必须把后面的 **s/""/"/g** 改为更通用的 **s/\\(.)/$1/g**，或者对应语言中的代码。

VB.NET 的 CSV 处理

```
Imports System.Text.RegularExpressions
  ......
Dim FieldRegex as Regex = New Regex( _
      "(?:^|,)                                              " & _
      "(?:                                                  " & _
      "    (?#  要么是双引号字段 ...)                          " & _
      "   ""   (?#  字段起始双引号)                            " & _
      "    (   (?> [^""]+ | """" )* )                        " & _
      "   ""   (?#  字段结束双引号)                            " & _
      "  (?# ... or ...)                                    " & _
      "  |                                                  " & _
      "    (?#  ... 非引号非逗号文本 ...)                      " & _
      "   ( [^"",]*)                                        " & _
      "  )", RegexOptions.IgnorePatternWhitespace)
Dim QuotesRegex as Regex = New Regex(" "" "" ")     '  双引号字符串
        .
        .
        .
Dim FieldMatch as Match = FieldRegex.Match(Line)
While FieldMatch.Success
   Dim Field as String
   If FieldMatch.Groups(1).Success
     Field = QuotesRegex.Replace(FieldMatch.Groups(1).Value, """")
   Else
     Field = FieldMatch.Groups(2).Value
   End If

   Console.WriteLine("[" & Field & "]")
   ' 现在可以处理'Field'

   FieldMatch = FieldMatch.NextMatch
End While
```



第6章 打造高效正则表达式

Crafting an Efficient Expression

Perl、Java、.NET、Python 和 PHP（这里没有列全，其他语言请参考第 145 页的表格）使用的都是表达式主导的 NFA 引擎，细微的改变就可能对匹配的结果及方式产生重大的影响。DFA 中不存在的问题，对 NFA 来说却很重要。因为 NFA 引擎容许用户进行精确控制，所以我们可以**用心打造**（译注 1）正则表达式，但对不熟悉的人来说，这样可能会带来麻烦。本章讲解的就是调校正则表达式的诀窍。

调校表达式时需要考虑的两个因素是准确性和效率：精确匹配我们需要的文本，不包含多余的内容，而且速度要快。第 4 章和第 5 章探讨了准确性，现在我们来考察 NFA 引擎的效率，以及如何有效地利用这些知识（我们会在合适的时候提到 DFA 的问题，不过本章主要关注的还是基于 NFA 的引擎）。总的来说，关键在于彻底理解回溯背后的过程，学习些技巧来避免可能的回溯。在深入了解了处理机制的细节之后，读者不但能够把匹配的速度提到最高，写更复杂的正则表达式时也会更有信心。

本章内容

为了让读者彻底掌握这些知识，本章首先说明了效率的重要性，然后回顾前几章讲解过的回溯，重点强调效率和回溯对整个匹配的影响，为掌握高级的技巧做准备。然后我们会考察一些常见的内部优化措施，它们可能对效率有相当程度的实质性影响；还要讲解，针对具体实现方式，如何构建最合适的正则表达式。最后，我会做个总结，传授一些终极技巧，来构建快如雷霆的 NFA 表达式。

译注 1：此处 craft 翻译为“打造”，下文中有时翻译为“调校”。

测试与回溯

我们将看到的例子代表了使用正则表达式时经常遇到的情况。在分析某个的正则表达式的效率时，我有时会列出正则引擎在匹配过程中进行的独立测试（individual test）的次数。如果用正则表达式「marty」匹配 smarty，一共会进行 6 次独立测试，首先是「m」对 s（匹配失败），然后是「m」对 m，「a」对 a，依次继续。我通常会给出回溯的次数（本例中回溯次数为 0，不过，正则引擎的传动装置必然会在第二个字符处重试正则表达式，这或许可以算作一次回溯）

之所以要列出这些数字，并不是为表明精确性，而是因为它们比“许多”、“少量”、“多次”、“更好”、“不太多”之类更为准确。我的意思不是说，在 NFA 上使用正则表达式需要精确地考察测试和回溯的次数，我只是希望让读者知道这些例子的相对优劣。

另一个重要的问题是，你必须意识到这些“精确”的数字可能根据工具的不同而有所不同。我期望读者能够知道，它只是针对具体例子的、相对的粗略表现。不同工具之间的一个重要区别就是，它们可能使用的优化措施不同。如果能够预先判断目标字符串基本无法匹配（例如目标字符串缺少一个引擎能够预知的，匹配成功必须的字符），足够聪明的实现方式可以完全不应用正则表达式。我在本章讨论了这些重要的优化措施，不过普遍原理比具体问题更为重要。

传统型 NFA 还是 POSIX NFA

在分析效率时，一定不要忘记所使用工具的引擎类型：传统型 NFA 还是 POSIX NFA。下一节中我们会看到，有些问题只对某种引擎存在。有的改变可能对其中之一没有影响，对另一个却有极大的影响。还是那句话，理解基本原理，就能应付各种情况。

典型示例

A Sobering Example

首先来看一个真正体现回溯和效率的重要性的例子。在 198 页，我们用「"(\\.|[^\\"])*"」来匹配引号字符串，其中容许出现转义的双引号。这个表达式没有错，但如果我们使用 NFA 引擎，对每个字符都应用多选结构的效率就会很低。对字符串中每个“正常”（非转义、非引用）的字符来说，这个引擎需要测试「\\.」，遇到失败后回溯，最终由「[^\\"]」匹配。如果效率不容忽视，就应该做些改动来加快匹配速度。

稍加修改——先迈最好使的腿

A Simple Change—Placing Your Best Foot Forward

对于一般的双引号字符串来说，普通字符的数量比转义字符要多，一个简单的改动就是调换两个多选分支的顺序，把「[^\\"]」放到「\\.」之前。这样，只有在遇到字符串中的转义字符时才会按照多选结构进行回溯（还有一次回溯是星号无法匹配引起的，此时所有的多选分支都匹配失败，所以整个多选结构无法匹配）。图 6-1 说明了其中的差异。箭头数量的减少，说明第一个多选分支的成功匹配次数增加了，也就是说回溯的次数减少了。

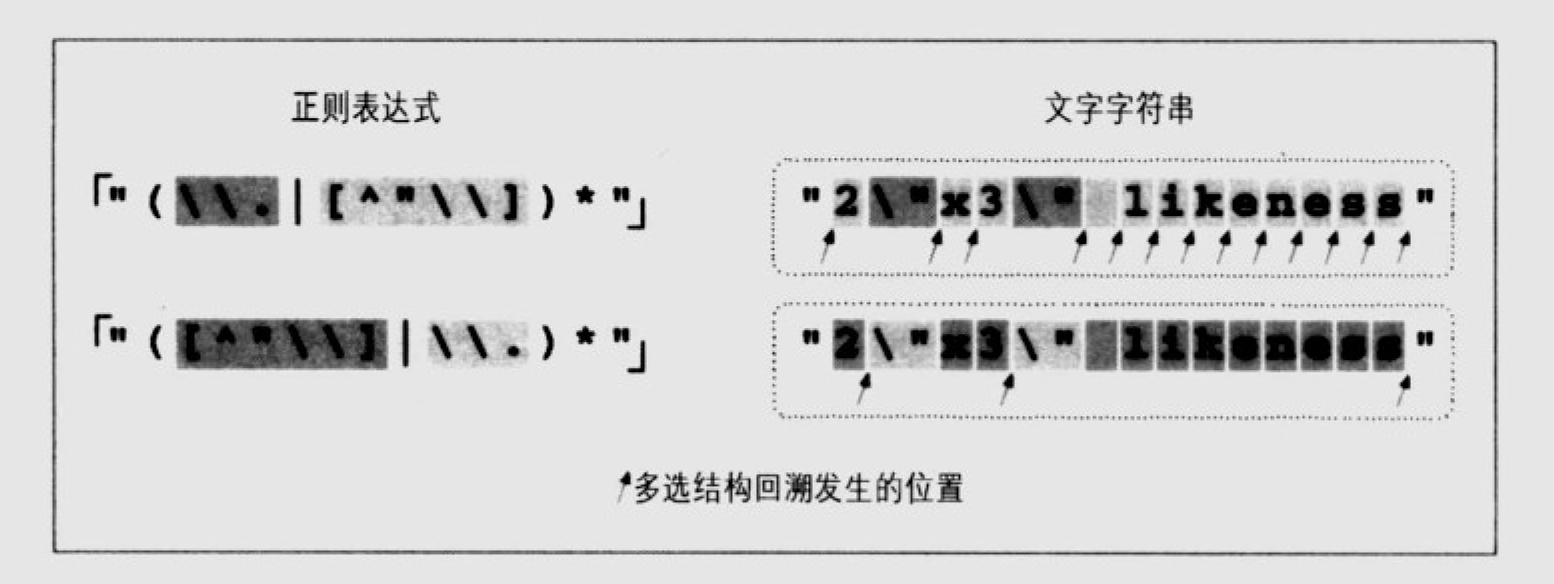


图 6-1：多选分支排列顺序的影响（传统型 NFA）

请从下面几个方面评价这个修改：

- 哪种引擎从中获益？传统型 NFA，或者 POSIX NFA，或是两者？
- 什么情况下，这种修改带来的收益最大？在文本能够匹配时，无法匹配时，还是所有时候。

❖ 请思考这些问题，翻到下一页查看答案。在阅读下一节以前，务必理解答案（及原因）。

效率 vs 准确性

Efficiency Versus Correctness

为提高效率修改正则表达式时最需要考虑的问题是，改动是否会影响匹配的准确性。像上面那样重新安排多选分支的顺序，只有在排序与匹配成功无关时才不会影响准确性。前一章出现的「"(\\.|[^"])*"」（☞197）的例子是有缺陷的。如果正则表达式只**需要**（should）应用于格式正确的字符串，此问题永远也不会暴露出来。如果认为这个表达式很不错，改

稍加改动的效果

❖ **223 页问题的答案**

哪种引擎从中获益？ 这种改动对 POSIX NFA 没有影响。因为它最终必须尝试正则表达式的每一种可能，多选分支的顺序其实不重要。不过，对传统型 NFA 来说，这样提高速度的多选分支重排序是有利的，因为引擎一旦找到匹配结果就会停下来。

什么样的情况下会有效果？ 只有匹配成功时才会加快速度。只有在尝试所有的可能（再说一次，POSIX NFA 任何情况下都会尝试所有可能）之后，NFA 才可能失败。所以如果确实不能匹配，每种可能都会被尝试，所以排列顺序没有影响。

下表列出了若干种情况下所进行的测试和回溯的次数（数字越小越好）：

目标字符串	传统型 NFA				POSIX NFA	
	「"(\\.|[^\\"])*"」		「"([^\\."]|\\.")*"」		两个表达式情况相同	
	测试	回溯	测试	回溯	测试	回溯
"2\"x3\" likeness"	32	14	22	4	48	30
"makudonarudo"	28	14	16	2	40	26
"very…*99 more chars*…long"	218	109	111	2	325	216
"No \"match\" here	124	86	124	86	124	86

我们发现，两个表达式在 POSIX NFA 中的情况是一样的，而修改之后，传统型 NFA 的表现提升了（减少了回溯）。而在不能匹配的情况下（最后一行），因为两种引擎必须尝试所有的可能，结果就是一样的。

动的确提高了效率，我们就会遇到真正的问题。交换多选分支，把「[^"]」放在前面，避免表达式进行不正确的匹配，如果目标字符串包含一个转义的双引号：

```
"You need a 2\"3\" photo."
```

所以，在关注效率的时候，万不可忘记准确性。

继续前进——限制匹配优先的作用范围

Advancing Further—Localizing the Greediness

从图 6-1 可以看出，在任意正则表达式中，星号会对每个普通字符进行迭代（或者说“重复”），重复进入-退出多选结构（和括号）。这需要成本，也就是额外的处理——如果可能，我们必须避免这些额外处理。

有一次，在处理这类正则表达式时，我想到一个优化的办法，考虑到「[^\\"]」匹配“普通”（非引号，非反斜线）的情况，使用「[^\\"]+」会在(…)*的一次迭代中读入尽可能多的字符。对没有转义字符的字符串来说，这样会一次读入整个字符串。于是就几乎不会进行回溯，也就把星号迭代的次数减少到最小。我很为自己的发现而高兴。

我们会在本章更深入地考察这个例子，不过看一眼统计数据会清楚地发现好处。图 6-2 展示了传统型 NFA 上应用这个例子的情况。比较原来的「"(\\.|[^\\"])*"」（上面的两个表达式），与多选结构相关的回溯和星号迭代都减少了。下面的两个例子说明，结合之前的重排序技巧，这种修改会带来更多的收益。



图 6-2：添加加号的结果（传统型 NFA）

新增的加号大大减少了多选结构回溯的次数，以及星号的迭代次数。星号量词作用于括号内的子表达式，每次迭代都需要进入然后再退出括号，这都需要成本，因为引擎需要记录

括号内的子表达式匹配的文本（本章会深入探讨此问题）。

表 6-1 与第 224 页答案中的表格类似，不过表达式不同，另外还给出了星号的迭代次数。在每种情况下，独立测试次数和回溯次数的增加都很有限，但是迭代次数有了显著降低，这是很大的进步。

表 6-1：传统型 NFA 的匹配效率

字符串	「"([^\\"]|\\.)*"」			「"([^\\"]+|\\.)*"」		
	测试	回溯	*迭代	测试	回溯	*迭代
"makudonarudo"	16	2	13	17	3	2
"2\"x3\" likeness"	22	4	15	25	7	6
"very...99 *more chars*...long"	111	2	108	112	3	2

实测

Reality Check

是的，我对自己的发现颇为得意。但这个看起来很奇妙的“改动”不过是场还未爆发的灾难。你可能注意到了，在考察它的各项指标时，我没有给出 POSIX NFA 的统计数据。如果这样做，你可能会很惊奇地发现"very·····long"的匹配需要超过 **3 亿亿亿**次（实际上是 324 518 553 658 426 726 783 156 020 576 256）回溯。说简单点就是，回溯是个天文数字。这需要超过 50 **百亿亿**（quintillion）年，或者是若干千万亿个千年（注 1）。

确实很出乎意料！那么，为什么会发生这种情况呢？简单地说，原因在于——这个正则表达式中某个元素受加号限定的同时，还受括号外的星号限定，无法区分哪个量词控制哪个特殊的字符。这种不确定性就是症结。下一节给出了详细解释。

“指数级”匹配

没有添加星号时，「[^\\"]」是星号的约束对象，真正的「([^\\"])*」能够匹配的字符是有限的。它先匹配一个字符，然后匹配下一个字符，如此继续，最多就是匹配目标文本中的每个字符。它也可能无法匹配目标字符串中的所有字符，不过，充其量，匹配字符的个数与目标字符串的长度成线性关系。目标字符串越长，可能的工作量相对也越大。

注 1：这个数字是根据其他测试估算出来的，我可没有那么长的时间来测试。

但是，对正则表达式的「([^\\"]+)*」来说，加号和星号二者分割（divvy up）字符串的可能性是成指数形式增长的。如果目标字符串是 makudonarudo，是星号会迭代 12 次，每一次迭代中「[^\\"]+」匹配一个字符（就像这样‘makudonarudo’）？还是星号迭代 3 次，内部的「[^\\"]+」分别匹配 5、3、4 个字符（‘makudonarudo’）？或者 2、2、5、3 个字符（‘makudonarudo’）？还是其他……

你现在知道，存在许多种可能（对长度为 12 的字符串存在 4096 种可能）。字符串中的每个字符，都存在两种可能，POSIX NFA 在给出结果之前必须尝试所有可能。这就是“指数级匹配”的来历。我还听说过一个不错的名字：“超线性（super-linear）”。

无论叫什么名字，终归都是回溯，大量的回溯（注 2）！12 个字符需要 4 096 种可能，这可能不需要多久时间，不过 20 个字符需要超过一百万种可能，时间长达若干秒。30 个字符，就需要超过十亿种可能，长达若干小时，如果是 40 个字符，就需要一年多的时间。这显然不是什么好事情。

你可能会想，“没关系，POSIX NFA 并不常见。我知道我的工具用的是传统型 NFA，所以这问题对我不存在。”的确，POSIX NFA 和传统型 NFA 的主要差别在于，传统型 NFA 在遇到第一个完整匹配可能时会停止。如果没有完整匹配，即使是传统型 NFA 也需要尝试所有的可能，在找到之前。即使是前面提到的"No·\"match\"·here 这样短短的字符串，在报告失败之前，也需要尝试 8 192 种可能。

在正则引擎忙于尝试这些数量庞大的可能时，整个程序看起来好像“锁死（lock up）”了。我第一次遇到这种情况时，以为自己发现了程序的 bug，不过现在我理解了，现在我把这个正则表达式加入自己的正则表达式工具包里，用来测试引擎的类型。

- 如果其中的某个表达式，即使不能匹配，也能很快给出结果，那可能就是 DFA。
- 如果只有在能够匹配时才很快出结果，那就是传统型 NFA。
- 如果总是很慢，那就是 POSIX NFA。

第一个判断中我使用了“可能”这个单词，因为经过高级优化的 NFA 没准能检测并且避免这些指数级的无休止（neverending）匹配（详见本章后文☛250）。同样，我们会见到各种方法来改进或重写这些表达式，加快它们匹配或报错的速度。

注 2：给感兴趣的读者两个数字，长度为 n 的字符串，回溯的次数是 2^{n+1}，独立测试的次数为 $2^{n+1}+2^n$。

前面列出的几点表明，如果排除某些高级优化的影响，就能根据正则表达式的相对性能判断引擎的类型。这就是第 4 章中（☞146）我们能用某些正则表达式来“测试引擎的类型”的原因。

当然，不是每点改动都会带来像本例一样的灾难性后果，不过除非知道正则表达式的幕后原理，否则在实际运行之前永远不能判断后果。为此，本章考察了各种例子的效率和后果。不过，对许多事来说，牢固理解基本概念对深入学习是非常重要的；所以，在讲解指数级匹配之前，我们不妨仔细复习复习回溯。

全面考察回溯

A Global View of Backtracking

从局部来看，回溯就是倒退至未尝试的分支。这很容易理解，但是回溯对整个匹配影响并不容易理解。在本节，我们会详细考察回溯在匹配成功和不成功时的各种细节，尝试从中发掘出一些东西。

先来仔细看看前一章的几个例子，在 165 页，我们把「".*"」应用到下面的文本：

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

匹配过程如图 6-3 所示。

正则表达式会从字符串的起始位置开始依次尝试每个字符，但是因为开头的引号无法匹配，此后的字符也不能匹配，直到尝试进行到标记位置 **A**。接着尝试表达式的其他部分，但是传动装置（☞148）知道如果这种尝试不成功，整个表达式可以从下一个位置开始尝试。

然后「.*」匹配直到字符串末尾，此时点号无法匹配，所以星号停止迭代。因为「.*」匹配成功可以不需要任何字符，所以在此过程中引擎记录了 46 个状态供回溯。现在「.*」停止了，引擎从最后保存的状态开始回溯，在“…**anise**▴”处开始尝试「".*▴"」。

也就是说，我们在字符串的末尾尝试匹配表示结束的双引号。不过，在这里双引号同样无法匹配，所以尝试仍然失败。然后引擎继续回溯、尝试，结果同样是无法匹配。

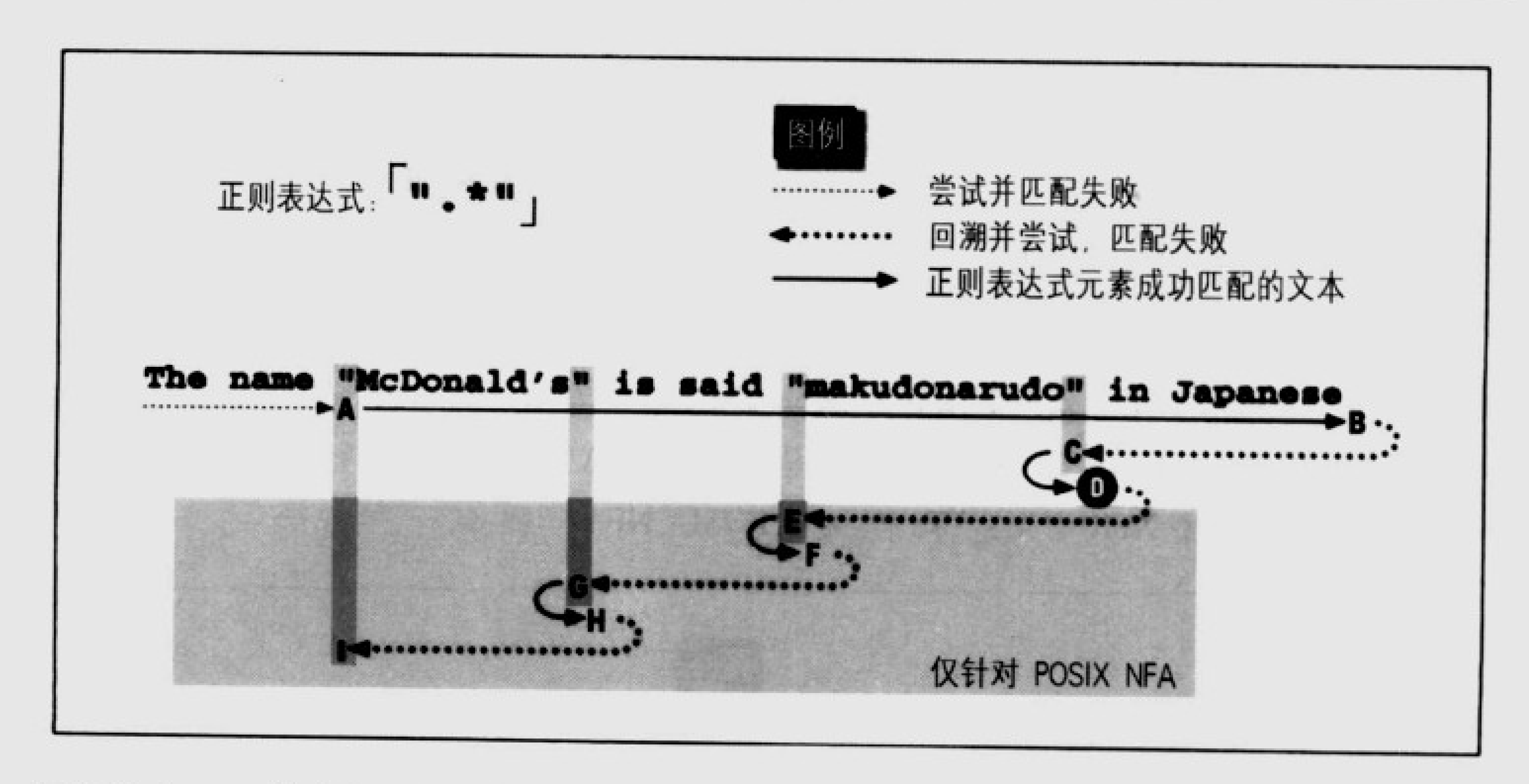


图 6-3：「".*"」的成功匹配过程

引擎倒过来尝试（最后保存的状态排在最先）从 **A** 到 **B** 保存的状态，首先是从 **B** 到 **C**。在进行了多次尝试之后到达这个状态：正则表达式中的「".*‸"」对应字符串中的…**arudo‸" ·in·Japa**…，也就是 **C** 所标注的位置。此时匹配成功，于是我们在 **D** 位置得到全局匹配。

```
The name "McDonald's" is said "Makudonarudo" in Japanese
```

这就是传统型 NFA 的匹配过程，剩下的未使用状态将被抛弃，报告匹配成功。

POSIX NFA 需要更多处理

More Work for a POSIX NFA

我们已经介绍过，POSIX NFA 的匹配是"到目前为止最长的匹配"，但是仍然需要尝试所有保存的状态，确认是否存在更长的匹配。我们知道，对本例来说，第一次找到的匹配就是最长的，但正则引擎需要确认这一点。

所以，在保存的所有状态中，除了两个能够匹配双引号的可能之外，其他都会在尝试后立即被放弃。所以，尝试过程 **D-E-F** 和 **F-G-H** 类似 **B-C-D**，只是 **F** 和 **H** 会被放弃，因为它们匹配的文本比 **D** 的要短。

在 **I** 位置能进行的回溯是"启动驱动过程，进行下一轮尝试（bump-along and retry）"。不过，因为从 **A** 位置开始的尝试能够找到匹配（实际上是三个），POSIX NFA 引擎最终停下来，报告在 **D** 位置的匹配。

无法匹配时必须进行的工作

Work Required During a Non-Match

我们还需要分析无法匹配时的情况。我们知道「".*"␣」无法匹配范例文本，但是它在匹配过程中仍然会进行许多工作。我们将会看到，工作量增大了许多。

图 6-4 说明了这些。**A-I** 序列类似图 6-3。区别在于，在位置 **D** 无法匹配（因为结尾的问号）。另一点区别在于，图 6-4 中的整个尝试序列是传统型 NFA 和 POSIX NFA 都必须经历的：如果无法匹配，传统型 NFA 必须进行的尝试与 POSIX NFA 一样多。

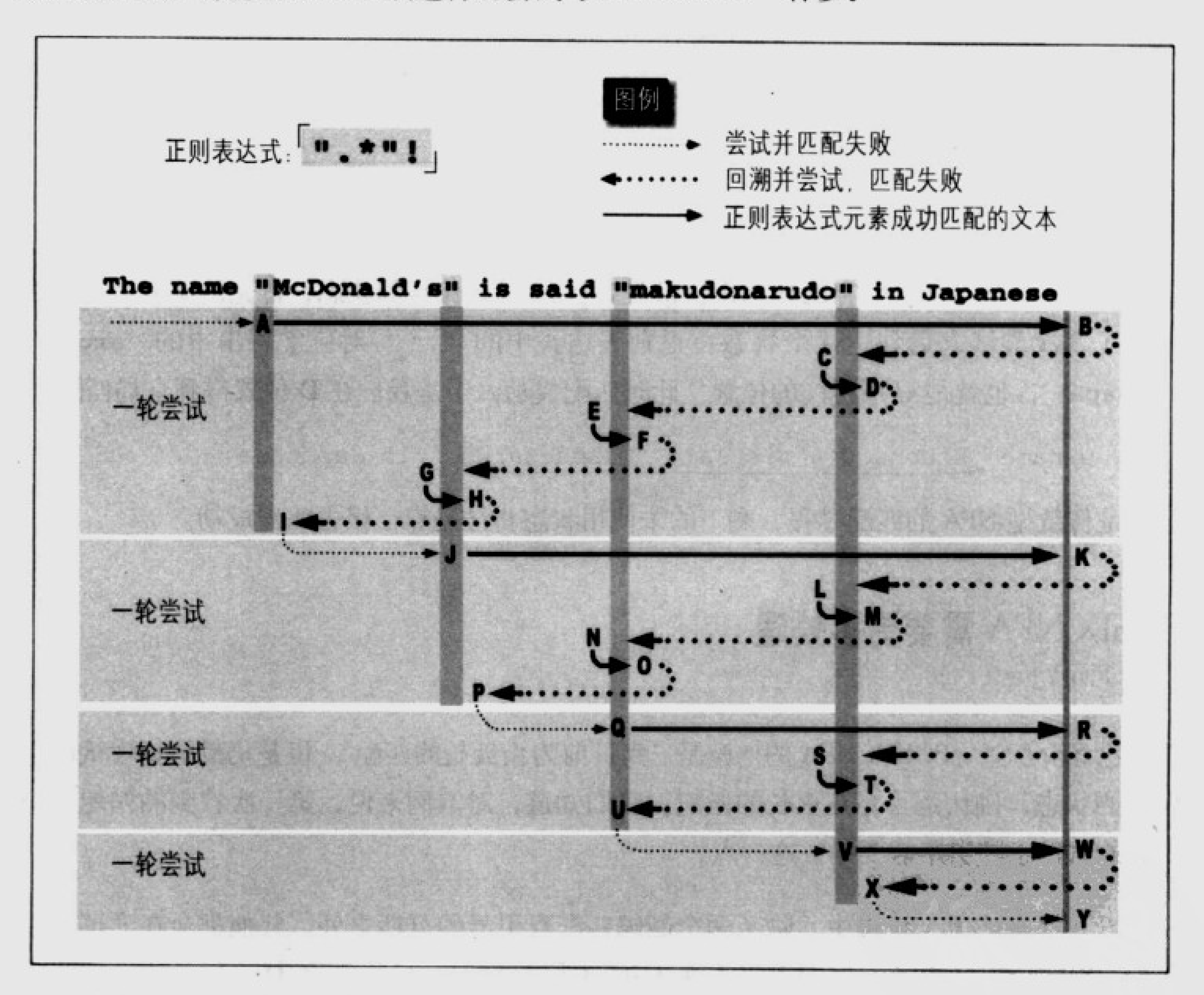


图 6-4：「".*"!」匹配失败的经过

因为从开始的 **A** 到结束的 **I** 的所有尝试都不存在匹配，传动装置必须启动驱动过程开始新一轮尝试。从 **J**、**Q**、**V** 开始的尝试看来有可能匹配成功，但结果都与从 **A** 开始的尝试一样。最终到 **Y**，不存在继续尝试的途径，所以整个尝试宣告失败。如图 6-4 所示，得到这个结果花费了许多工夫。

看清楚一点

Being More Specific

我们把点号换成「[^"]」来做个比较。前一章已经讨论过，这样的结果更容易理解，因为它能匹配的字符更少，而且这样一来，正则表达式的效率也提高了。如果使用「"[^"]*"!」，「[^"]*」匹配的内容就不能包括双引号，减少了匹配和回溯。

图 6-5 说明了尝试失败的过程（请对比图 6-4）。从图中可以看到，回溯的次数大大减少了。如果这个结果满足我们的需要，减少的回溯就是有益的伴随效应（side effect）。

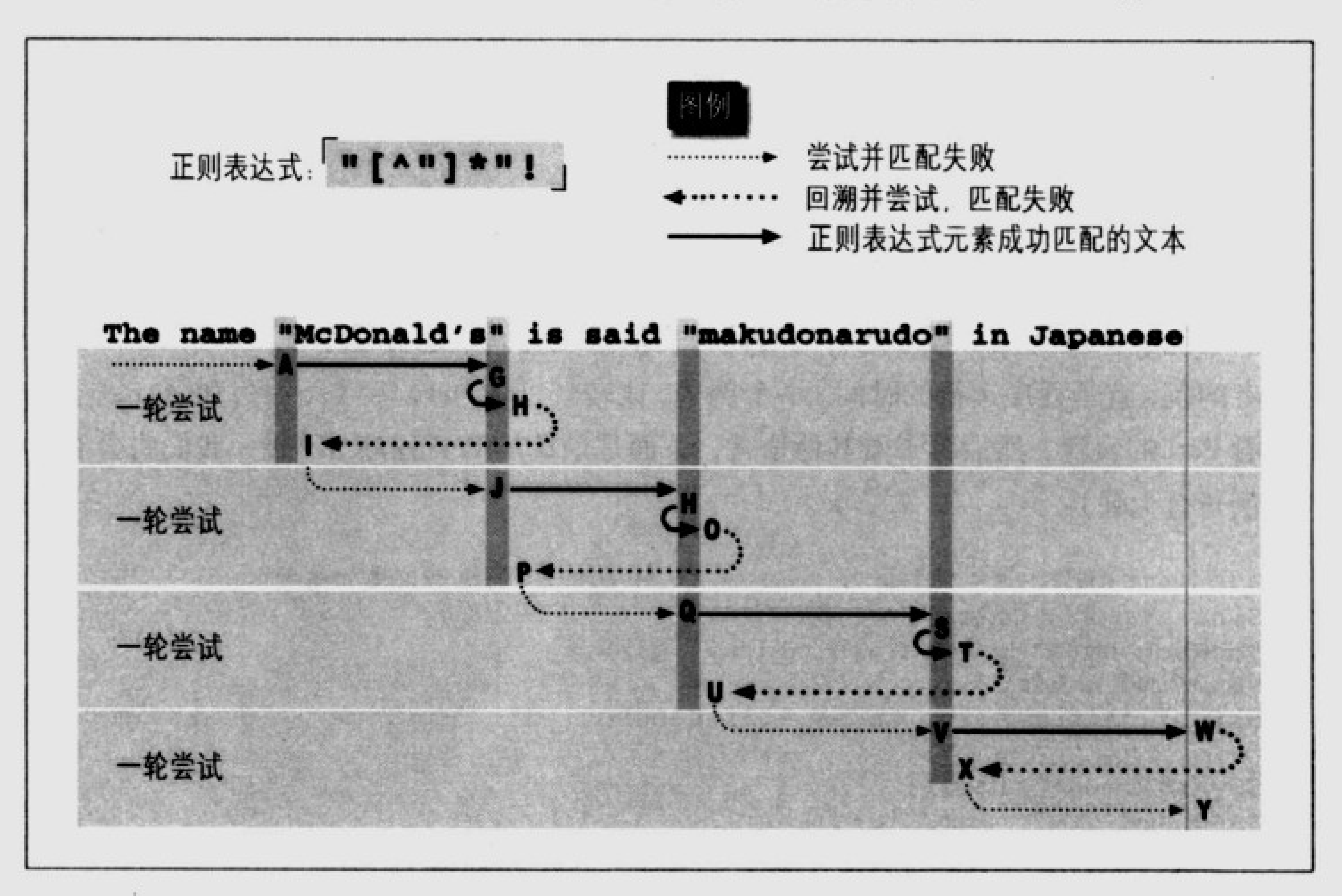


图 6-5：「"[^"]*"!」无法匹配

多选结构的代价很高

Alternation Can Be Expensive

多选结构或许是回溯的主要原因。举个简单的例子，用 makudonarudo 来比较「u|v|w|x|y|z」和「[uvwxyz]」的匹配。字符组一般只是进行简单测试（注 3），所以「[uvwxyz]」只需要进行 34 次尝试就能匹配。

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

注 3：不同实现方式的效率可能存在差异，但总的来说字符组的效率要比相应的多选结构高。

如果使用「u|v|w|x|y|z」，则需要在每个位置进行 6 次回溯，在得到同样结果之前总共有 204 次回溯。当然，并不是每个多选结构都可以替换为字符组，即使可以，也不见得会这么简单。不过，在某些情况下，我们将要学习的技巧能够大大减少与匹配所须的多选结构相关的回溯。

理解回溯可能是学习 NFA 效率中最重要的问题，但所有的问题不只于此。正则引擎的优化措施能够大大提升效率。在本章后面，我们将详细考察正则引擎要做的工作和优化手段。

性能测试

Benchmarking

本章主要讲解速度和效率，而且会时常使用性能测试，所以我希望介绍一些测试的原则。我会用几种语言来介绍简单的测试方法。

基本的性能测试就是记录程序运行的时间：先取系统时间，运行程序，再取系统时间，计算两者的差，就是程序运行的时间。举个例子，比较「^(a|b|c|d|e|f|g)+$」和「^[a-g]+$」。先来看 Perl 的表现，然后再来看其他语言。下面是简单的 Perl 程序（不过，我们将会看到，这个例子有欠缺）：

```
use Time::HiRes  'time';       #  这样 time() 的返回值更加精确
$StartTime = time();
"abababdedfg" =~ m/^(a|b|c|d|e|f|g)+$/;
$EndTime = time();
printf("Alternation takes %.3f seconds.\n", $EndTime - $StartTime);

$StartTime = time();
"abababdedfg" =~ m/^[a-g]+$/;
$EndTime = time();
printf("Character class takes %.3f seconds.\n", $EndTime - $StartTime);
```

它看来（而且也确实是）很简单，但是在进行性能测试时，我们需要记住几点：

- **只记录"真正关心的（interesting）"处理时间。**尽可能准确地记录"处理"时间，尽可能避免"非处理时间"的影响。如果在开始前必须进行初始化或其他准备工作，请在它们完成之后开始计时；如果需要收尾工作，请在计时停止之后进行这些工作。

- **进行"足够多"的处理。**通常，测试需要的时间是相当短暂的，而计算机时钟的单位精度不够，无法给出有意义的数值。

在我的机器上运行这个 Perl 程序，结果是：

```
Alternation takes 0.000 seconds.
Character class takes 0.000 seconds.
```

我们只能知道，这段程序所需的时间比计算机能够测量的最短时间还要短。所以，如果程序运行的时间太短，就运行两次、十次，甚至一千万次，来保证“足够多”的工作。这里的“足够多”取决于系统时钟的精度，大多数系统能够精确到 1/100s，这样，即使程序只需要 0.5s，也能取得有意义的结果。

- **进行“准确的”处理**。进行 1 000 万次快速操作需要在负责计时的代码块中升级 1 000 万次计数器。如果可能，最好的办法是增加真正的处理部分的比例，而不增加额外的开销。在 Perl 的例子中，正则表达式应用的文本相当短：如果应用到长得多的字符串，在每次循环中所作的“真正的”处理也会多一些。

考虑到这些因素，我们可以得出下面的程序：

```
use Time::HiRes 'time';                # 这样 time()的返回值更加精确
$TimesToDo = 1000;                     # 设定重复次数
$TestString = "abababdedfg" x 1000;    # 生成长字符串

$Count = $TimesToDo;
$StartTime = time();
while ($Count-- > 0) {
    $TestString =~ m/^(a|b|c|d|e|f|g)+$/;
}
$EndTime = time();
printf("Alternation takes %.3f seconds.\n", $EndTime - $StartTime);

$Count = $TimesToDo;
$StartTime = time();
while ($Count-- > 0) {
    $TestString =~ m/^[a-g]+$/;
}
$EndTime = time();
printf("Character class takes %.3f seconds.\n", $EndTime - $StartTime);
```

请注意，`$TestString` 和`$Count` 的初始化在计时开始之前（`$TestString` 使用了 Perl 提供的 `x` 操作符进行初始化，它表示将左边的字符串重复右边的次数）。在我的机器上，使用 Perl5.8 运行的结果是：

```
Alternation takes 7.276 seconds.
Character class takes 0.333 seconds.
```

所以，对这个例子来说，多选结构要比字符组快 22 倍左右。此测试应该执行多次，选取最短的时间，以减少后台系统活动的影响。

理解测量对象

Know What You're Measuring

我们把初始化程序更改为下面这样，会得到更有意思的结果：

```
$TimesToDo = 1000000;
$TestString = "abababdedfg";
```

现在，测试字符串只是上面的长度的 1/1000，而测试需要进行 1000 次。每个正则表达式测试和匹配的字符总数并没有变化，因此从理论上讲，“工作量”应该没有变化。不过，结果却大不相同：

```
Alternation takes 18.167 seconds.
Character class takes 5.231 seconds.
```

两个时间都比之前的要长。原因是新增的“非处理”开销——对`$Count`的检测和更新，以及建立正则引擎的时间，现在的次数是以前的 1 000 倍。

对于字符组测试来说，新增的开销花费了大约 5s 的时间，而多选结构则增加了将近 10 秒。为什么多选结构测试的时间变化如此之大？主要是因为捕获型括号（在每次测试之前和之后，它们都需要额外处理，这样的操作要多 1 000 倍）。

无论如何，进行这点修改的要点在于说明，真正处理部分和非真正处理部分在计时中所占的比重会强烈地影响到测试结果。

PHP 测试

Benchmarking with PHP

下面是 PHP 的测试，使用 preg 引擎：

```
$TimesToDo = 1000;

/* 准备测试字符串 */
$TestString = "";
for ($i = 0; $i < 1000; $i++)
   $TestString .= "abababdedfg";

/* 开始第一轮测试 */
$start = gettimeofday();
for ($i = 0; $i < $TimesToDo; $i++)
    preg_match('/^(a|b|c|e|f|g)+$/', $TestString);
$final = gettimeofday();
$sec = ($final['sec'] + $final['usec']/1000000) -
      ($start['sec'] + $start['usec']/1000000);
printf("Alternation takes %.3f seconds\n", $sec);

/* 开始第二轮测试 */
$start = gettimeofday();
for ($i = 0; $i < $TimesToDo; $i++)
    preg_match('/^[a-g]+$/', $TestString);
$final = gettimeofday();
$sec = ($final['sec'] + $final['usec']/1000000) -
      ($start['sec'] + $start['usec']/1000000);
printf("Character class takes %.3f seconds\n", $sec);
```

在我的机器上，结果是：

```
Alternation takes 27.404 seconds
Character class takes 0.288 seconds
```

如果在测试中遇到 PHP 错误“not being safe to rely on the system's timezone settings”，请添加下面的代码：

```
if (phpversion() >= 5)
   date_default_timezone_set("GMT");
```

Java 测试

Benchmarking with Java

因为某些原因，用 Java 测试很有讲究。首先看个考虑不够周到的例子，然后请思考它为什么考虑不周到，应该如何改进：

```
import java.util.regex.*;
public class JavaBenchmark {
public static void main(String [] args)
{
   Matcher regex1 = Pattern.compile("^(a|b|c|d|e|f|g)+$").matcher("");
   Matcher regex2 = Pattern.compile("^[a-g]+$").matcher("");
   long timesToDo = 1000;

   StringBuffer temp = new StringBuffer();
   for (int i = 1000; i > 0; i--)
         temp.append("ababab dedfg");
   String testString = temp.toString();

   // 第一轮测试计时 ...
   long count = timesToDo;
   long startTime = System.currentTimeMillis();
   while (--count > 0)
        regex1.reset(testString).find();
   double seconds = (System.currentTimeMillis() - startTime)/1000.0;
   System.out.println("Alternation takes " + seconds + " seconds");

  // 第二轮测试计时 ...
  count = timesToDo;
  startTime = System.currentTimeMillis();
  while (--count > 0)
       regex2.reset(testString).find();
  seconds = (System.currentTimeMillis() - startTime)/1000.0;
  System.out.println("Character class takes " + seconds + " seconds");
  }
}
```

你注意到在这个程序中正则表达式如何初始化部分编译了吗？我们需要测试的是匹配的速度，而不是编译的速度。

速度取决于所使用的虚拟机（VM）。Sun 的标准 JRE 有两种虚拟机，*client* VM 为快速启动而优化，*server* VM 为长时间、大负荷的作业而优化。

在我的机器上，使用 client VM 运行测试的结果如下：

```
Alternation takes 19.318 seconds
Character class takes 1.685 seconds
```

使用 server VM 的结果如下：

```
Alternation takes 12.106 seconds
Character class takes 0.657 seconds
```

这样看来测试有点不可信了，之所以说它不够周到，原因在于计时的结果在很大程度上取决于自动的预执行编译器（automatic pre-execution compiler）的工作，或者说运行时编译器（run-time compiler）与测试代码的交互情况。某些虚拟机包含 JIT（Just-In-Time compiler），JIT 会根据需要，在需要执行代码之前才进行编译。

Java 使用了我称为 BLTN（Better-Late-Than-Never）的编译器，在执行期间计数，对反复使用的代码根据需要进行编译和优化。BLTN 的性质是，它只对认为“热门”（hot，即大量使用）的代码进行干预。如果虚拟机已经运行了一段时间，例如在服务器环境中，它已经“预热”完毕，而我们的简单例子确保了一台“凉”的服务器（BLTN 没有进行任何优化）。

可以把测试部分放入一个循环，来观察“预热”现象：

```
// 第一轮测试计时 ...
for (int i = 4; i > 0; i--)
{
    long count = timesToDo;
    long startTime = System.currentTimeMillis();
    while (--count > 0)
         regex1.reset(testString).find();
    double seconds = (System.currentTimeMillis() - startTime)/1000.0;
    System.out.println("Alternation takes " + seconds + " seconds");
}
```

如果新增的循环运行足够长（例如，10s），BLTN 就会优化热门代码，最后一次输出的时间就代表了已预热系统的情况。再次使用 server VM，这些时间确实比之前有了 8%和 25%的提高。

```
Alternation takes 11.151 seconds
Character class takes 0.483 seconds
```

另一个问题在于，负责调度 GC 线程的工作是不确定的。所以，进行足够长时间的测试能够降低这些不确定因素的影响。

VB.NET 测试

Benchmarking with VB.NET

下面是 VB.NET 的测试程序：

```
Option Explicit On
Option Strict On

Imports System.Text.RegularExpressions

Module Benchmark
Sub Main()
  Dim Regex1 as Regex = New Regex("^(a|b|c|d|e|f|g)+$")
  Dim Regex2 as Regex = New Regex("^[a-g]+$")
  Dim TimesToDo as Integer = 1000
  Dim TestString as String = ""
  Dim I as Integer
  For I = 1 to 1000
     TestString = TestString & "abababdedfg"
  Next

  Dim StartTime as Double = Timer()
  For I = 1 to TimesToDo
     Regex1.Match(TestString)
  Next
  Dim Seconds as Double = Math.Round(Timer() - StartTime, 3)
  Console.WriteLine("Alternation takes " & Seconds & " seconds")

  StartTime = Timer()
  For I = 1 to TimesToDo
     Regex2.Match(TestString)
  Next
  Seconds = Math.Round(Timer() - StartTime, 3)
  Console.WriteLine("Character class takes " & Seconds & " seconds")
End Sub
End Module
```

在我的机器上，结果是：

```
Alternation takes 13.311 seconds
Character class takes 1.680 seconds
```

在.NET Framework 中使用 `RegexOptions.Compiled` 作为正则表达式构造函数的第 2 个参数，能够把正则表达式编译为效率更高的形式（☞410），其结果为：

```
Alternation takes 5.499 seconds
Character class takes 1.157 seconds
```

使用 `Compiled` 功能之后，两个测试的速度都有提高，但是多选结构的相对上升幅度更为明显（几乎是之前的 3 倍，而字符组的程序只提高到之前的 1.5 倍），所以多选结构从中获益更大。

Ruby 测试

Benchmarking with Ruby

下面是 Ruby 的测试代码：

```
TimesToDo=1000
testString=""
for i in 1..1000
   testString += "abababdedfg"
end
Regex1 = Regexp::new("^(a|b|c|d|e|f|g)+$");
Regex2 = Regexp::new("^[a-g]+$");
startTime = Time.new.to_f
for i in 1..TimesToDo
   Regex1.match(testString)
end
print "Alternation takes %.3f seconds\n" % (Time.new.to_f - startTime);
startTime = Time.new.to_f
for i in 1..TimesToDo
   Regex2.match(testString)
end
print "Character class takes %.3f seconds\n" % (Time.new.to_f - startTime);
```

在我的机器上，结果如下：

```
Alternation takes 16.311 seconds
Character class takes 3.479 seconds
```

Python 测试

Benchmarking with Python

下面是 Python 的测试代码：

```
import re
import time
import fpformat
Regex1 = re.compile("^(a|b|c|d|e|f|g)+$")
Regex2 = re.compile("^[a-g]+$")
TimesToDo = 1250;
TestString = ""
for i in range(800):
   TestString += "abababdedfg"
StartTime = time.time()
for i in range(TimesToDo):
   Regex1.search(TestString)
Seconds = time.time() - StartTime
print "Alternation takes " + fpformat.fix(Seconds,3) + " seconds"
StartTime = time.time()
for i in range(TimesToDo):
   Regex2.search(TestString)
Seconds = time.time() - StartTime
print "Character class takes " + fpformat.fix(Seconds,3) + " seconds"
```

因为 Python 的正则引擎设定的限制，我们必须减少字符串的长度，因为原来长度的字符串会导致内部错误（“maximum recursion limit exceeded”）。这种规定有点像减压阀，它有助于终止无休止匹配。

作为弥补，我相应增加了测试的次数。在我的机器上，测试结果为：

```
Alternation takes 10.357 seconds
Character class takes 0.769 seconds
```

Tcl 测试

Benchmarking with Tcl

下面是 Tcl 的测试代码：

```
set TimesToDo 1000
set TestString ""
for {set i 1000} {$i > 0} {incr i -1} {
    append TestString "abababdedfg"
}

set Count $TimesToDo
set StartTime [clock clicks -milliseconds]
for {} {$Count > 0} {incr Count -1} {
    regexp {^(a|b|c|d|e|f|g)+$} $TestString
}
set EndTime [clock clicks -milliseconds]
set Seconds [expr ($EndTime - $StartTime)/1000.0]
puts [format "Alternation takes %.3f seconds" $Seconds]

set Count $TimesToDo
set StartTime [clock clicks -milliseconds]
for {} {$Count > 0} {incr Count -1} {
    regexp {^[a-g]+$} $TestString
}
set EndTime [clock clicks -milliseconds]
set Seconds [expr ($EndTime - $StartTime)/1000.0]
puts [format "Character class takes %.3f seconds" $Seconds]
```

在我的机器上，结果如下：

```
Alternation takes 0.362 seconds
Character class takes 0.352 seconds
```

神奇的是，两者速度相当。还记得吗，我们在第 145 页说过，Tcl 使用的是 NFA/DFA 混合引擎，对 DFA 引擎来说，这两个表达式是没有区别的。本章所举的大部分例子并不适用于 Tcl，详细信息请参考第 243 页。

常见优化措施

Common Optimizations

聪明的正则表达式实现（implementation）有许多办法来优化，提高取得结果的速度。优化通常有两种办法：

- **加速某些操作**。某些类型的匹配，例如「\d+」，极为常见，引擎可能对此有特殊的处理方案，执行速度比通用的处理机制要快。
- **避免冗余操作**。如果引擎认为，对于产生正确结果来说，某些特殊的操作是不必要的，或者某些操作能够应用到比之前更少的文本，忽略这些操作能够节省时间。例如，一个以「\A」（行开头）开头的正则表达式只有在字符串的开头位置才能匹配，如果在此处无法匹配，传动装置不会徒劳地尝试其他位置（进行无谓的尝试）。

在下面的十几页中，我会讲解自己见过的许多种不同的天才优化措施。没有任何一种语言或者工具提供了所有这些措施，或者只是与其他语言和工具相同的优化措施，我也确信，还有许多我没见过的优化措施，但看完本章的读者，应该能够合理利用自己所用工具提供的任何优化措施。

有得必有失

No Free Lunch

通常来说优化能节省时间，但并非永远如此。只有在检测优化措施是否可行所需的时间少于节省下来的匹配时间的情况下，优化才是有益的。事实上，如果引擎检查之后认为不可能进行优化，结果总是会更慢，因为最开始的检查需要花费时间。所以，在优化所需的时间，节省的时间，以及更重要的因素——优化的可能性之间，存在互相制约的关系。

来看一个例子。表达式「\b\B」（某个位置既是**单词分隔符**又**不是单词分隔符**）是不可能匹配的。如果引擎发现提供的表达式包含「\b\B」就知道整个表达式都无法匹配，因而不会进行任何匹配操作。它会立刻报告匹配失败。如果匹配的文本很长，节省的时间就非常可观。

不过，我所知的正则引擎都没有进行这样的优化。为什么？首先，很难判断这条规则是否适用于某个特定的表达式。某个包含「\b\B」的正则表达式很可能可以匹配，所以引擎必须做一些额外的工作来预先确认。当然，节省的时间确实很可观，所以如果预计到「\b\B」经常出现，这样做还是值得的。

不幸的是，这并不常见（而且愚蠢）（注 4），虽然在极少数情况下这样做可以节省大量的时间，但其他情况下速度降低的代价比这高得多。

优化各有不同

Everyone's Lunch is Different

在讲解各种优化措施时，请务必记住一点“优化各有不同（everyone's lunch is different）”。虽然我尽量使用简单清晰的名字来命名每种措施，但不同的引擎必然可能以不同的方式来优化。对某个正则表达式进行细微的改动，在某个实现方式中可能会带来速度的大幅提升，而在另一个实现方式中大大降低速度。

正则表达式的应用原理

The Mechanics of Regex Application

我们必须先掌握正则表达式应用的基本知识，然后讲解先进系统的优化原理及利用方式。之前已经了解了回溯的细节，在本节我们要进行更全面地学习。

正则表达式应用到目标字符串的过程大致分为下面几步：

1. **正则表达式编译** 检查正则表达式的语法正确性，如果正确，就将其编译为内部形式（internal form）。
2. **传动开始** 传动装置将正则引擎“定位”到目标字符串的起始位置。
3. **元素检测** 引擎开始测试正则表达式和文本，依次测试正则表达式的各个元素（component），如第 4 章所说的那样。我们已经详细考察了 NFA 的回溯，但是还有几点需要补充：
   - 相连元素，例如「Subject」中的「S」、「u」、「b」、「j」、「e」等等，会依次尝试，只有当某个元素匹配失败时才会停止。
   - 量词修饰的元素，控制权在量词（检查量词是否应该继续匹配）和被限定的元素（测试能否匹配）之间轮换。
   - 控制权在捕获型括号内外进行切换会带来一些开销。括号内的表达式匹配的文本必须保留，这样才能通过$1 来引用。因为一对括号可能属于某个回溯分支，括号的状态就是用于回溯的状态的一部分，所以进入和退出捕获型括号时需要修改状态。

注 4：事实上，我曾在测试中使用「\b\B」来保证正则表达式的某个部分匹配失败。例如，我可能把「\b\B」插入「…(this␣|this other)…」标记的状态中，保证第一个多选分支的失败。现在，当我使用“必须失败”的元素时，我使用「(?!)」，你可以在第 333 页找到这个与 Perl 相关的例子。

4. **寻找匹配结果** 如果找到一个匹配结果，传统型 NFA 会“锁定”在当前状态，报告匹配成功。而对 POSIX NFA 来说，如果这个匹配是迄今为止最长的，它会记住这个可能的匹配，然后从可用的保存状态继续下去。保存的状态都测试完毕之后返回最长的匹配。

5. **传动装置的驱动过程** 如果没有找到匹配，传动装置就会驱动引擎，从文本中的下一个字符开始新一轮的尝试（回到步骤3）。

6. **匹配彻底失败** 如果从目标字符串的每一个字符（包括最后一个字符之后的位置）开始的尝试都失败了，就会报告匹配彻底失败。

下面几节讲解高级的实现方式如何减少这些处理，以及如何应用这些技巧。

应用之前的优化措施

Pre-Application Optimizations

优秀的正则引擎实现方式能够在正则表达式实际应用之前就进行优化，它有时候甚至能迅速判断出，某个正则表达式无论如何也无法匹配，所以根本不必应用这个表达式。

编译缓存

第2章的 E-mail 处理程序中，用于处理 header 各行的主循环体中是这样的：

```
while (…) {
    if ($line =~ m/^\s*$/ ) …
    if ($line =~ m/^Subject: (.*)/)…
    if ($line =~ m/^Date: (.*)/)…
    if ($line =~ m/^Reply-To: (\S+)/)…
    if ($line =~ m/^From: (\S+) \(([^()]*)\)/)…
}
```

正则表达式使用之前要做的第一件事情是进行错误检查，如果没有问题则编译为内部形式。编译之后的内部形式能用来检查各种字符串，但是这段程序的情况如何？显然，每次循环都要重新编译所有正则表达式，这很浪费时间。相反，在第一次编译之后就把内部形式保存或缓存下来，在此后的循环中重复使用它们，显然会提高速度（只是要消耗些内存）。

具体做法取决于应用程序提供的正则表达式处理方式。93页已经说过，有3种处理方式：集成式、程序式和面向对象式。

集成式处理中的编译缓存

Perl 和 awk 使用的就是集成式处理方法，非常容易进行编译缓存。从内部来说，每个正则表达式都关联到代码的某一部分，第一次执行时在编译结果与代码之间建立关联，下次执行时只需要引用即可。这样最节省时间，代价就是需要一部分内存来保存缓存的表达式。

DFA，Tcl 与手工调校正则表达式

本章中介绍的大部分优化措施并不适用于 DFA。第 242 页介绍的**编译缓存**却适用于所有的引擎，但是本章讨论的所有手工调校都不适用于 DFA。第 4 章已经澄清，逻辑上相等的正则表达式「`this|that`」和「`th(is|at)`」，对 DFA 来说是等价的。之所以要写本章，是因为它们对 NFA 来说不相等。

那么使用 DFA/NFA 混合引擎的 Tcl 呢？Tcl 的正则引擎是由正则表达式的传奇人物 Henry Spencer（☞88）为 Tcl 专门开发的，它成功地融合了 DFA 和 NFA 的优点。在 2000 年 4 月的 Usenet posting 中，Henry 这样写到：

> 总的来说，与传统的正则引擎相比，Tcl 的引擎对正则表达式的具体形式的敏感度要低得多。无论正则表达式写成怎么样，该快的就快，该慢的就慢。传统的正则表达式手工优化在这里不适用。

Henry 的 Tcl 正则引擎是一大进步。如果其中的技术流行开来，本章的许多内容就毫无意义了。

变量插值功能（variable interpolation，即将变量的值作为正则表达式的一部分）可能会给缓存造成麻烦。例如对 **m/**`^Subject:·\Q` **`$DesiredSubject`**`\E\s*$`**/** 来说，每次循环中正则表达式的内容可能会发生改变，因为它取决于插值变量，而这个变量的值可能会变化。如果每次都会不同，那么正则表达式每次都需要编译，完全不能重复利用。

尽管正则表达式**可能**每次循环都会变化，但这并不是说任何时候都需要重新编译。折中的优化措施就是检查插值后的结果（也就是正则表达式的具体值），只有当具体值发生变化时

才重新编译。不过，如果变化的几率很小，大多数时候就只需要检查（而不需要编译），优化效果很明显。

程序式处理中的编译缓存

在集成式处理中，正则表达式的使用与其在程序中所处的具体位置相关，所以再次执行这段代码时，编译好的正则表达式就能够缓存和重复使用。但是，程序式处理中只有通用的"应用此表达式"的函数。也就是说，编译形式并不与程序的具体位置相连，下次调用此函数时，正则表达式必须重新编译。从理论上来说就是如此，但是在实际应用中，禁止尝试缓存的效率无疑很低。相反，优化通常是把最近使用的正则表达式模式（regex pattern）保存下来，关联到最终的编译形式。

调用"应用此表达式"函数之后，作为参数的正则表达式模式会与保存的正则表达式相比较，如果存在于缓存中，就使用缓存的版本。如果没有，就直接编译这个正则表达式，将其存入缓存（如果缓存有容量限制，可能会替换一个旧的表达式）。如果缓存用完了，就必须放弃（thrown out）一个编译形式，通常是最久未使用的那个。

GNU Emacs 的缓存能够保存最多 20 个正则表达式，Tcl 能保存 30 个。PHP 能保存四千多个。.NET Framework 在默认情况下能保存 15 个表达式，不过数量可以动态设置，也可以禁止此功能（☞432）。

缓存的大小很重要，因为如果缓存装不下循环中用到的所有正则表达式，在循环重新开始时，最开始的正则表达式会被清除出缓存，结果每个正则表达式都需要重新编译。

面向对象式处理中的编译缓存

在面向对象式处理中，正则表达式何时编译完全由程序员决定。正则表达式的编译是用户通过 **New Regex**、**re.compile** 和 **Pattern.compile**（分别对应.NET、Python 和 `java.util.regex`）之类的构造函数来进行的。第 3 章的简单示例对此做了介绍（从第 95 页开始），编译在正则表达式实际应用之前完成，但是它们也可以更早完成（有时候可以在循环之前，或者是程序的初始化阶段），然后可以随意使用。在第 235、237 和 238 页的性能测试中体现了这一点。

在面向对象式处理中，程序员通过对象析构函数抛弃（thrown away）编译好的正则表达式。

及时抛弃不需要的编译形式能够节省内存。

预查必须字符/子字符串优化

相比正则表达式的完整应用，在字符串中搜索某个字符（或者是一串字符）是更加“轻量级”的操作，所以某些系统会在编译阶段做些额外的分析，判断是否存在成功匹配必须的字符或者字符串。在实际应用正则表达式之前，在目标字符串中快速扫描，检查所需的字符或者字符串——如果不存在，根本就不需要进行任何尝试。

举例来说，「^Subject:·(.*)」的‘Subject:·’是必须出现的。程序可以检查整个字符串，或者使用 *Boyer-Moore* 搜索算法（这是一种很快的文件检索算法，字符串越长，效率越高）。没有采用 Boyer-Moore 算法的程序进行逐个字符检查也可以提高效率。选择目标字符串中不太可能出现的字符（例如‘Subject:·’中的‘t’之后的‘:’）能够进一步提高效率。

正则引擎必须能识别出，「^Subject:·(.*)」的一部分是固定的文本字符串，对任意匹配来说，识别出「this|that|other」中‘th’是必须的，需要更多的工夫，而且大多数正则引擎不会这样做。此问题的答案并不是黑白分明的，某个实现方式或许不能识别出‘th’是必须的，但能够识别出‘h’和‘t’都是必须的，所以至少可以检查一个字符。

不同的应用程序能够识别出的必须字符和字符串有很大差别。许多系统会受到多选结构的干扰。在这种系统中，使用「th(is|at)」的表现好于「this|that」。同样，请参考第 247 页的“开头字符/字符组/子串识别优化”。

长度判断优化

「^Subject:·(.*)」能匹配文本的长度是不固定的，但是至少必须包含 9 个字符。所以，如果目标字符串的长度小于 9 则根本不必尝试。当然，需要匹配的字符更长优化的效果才更明显，例如「:\d{79}:」（至少需要 81 个字符）。

请参见第 247 页的“长度识别传动优化”。

通过传动装置进行优化

Optimizations with the Transmission

即使正则引擎无法预知某个字符串能否匹配，也能够减少传动装置真正应用正则表达式的位置。

字符串起始/行锚点优化

这种优化措施能够判断，任何以「^」开头的正则表达式只能在「^」能够匹配的情况下才可能匹配，所以只需要在这些位置应用即可。

在"预查必须字符/子字符串优化"中提到，正则引擎必须判断对某个正则表达式来说有哪些可行的优化，在这里同样有效。任何使用此优化的实现方式都必须能够识别：如果「^(this|that)」匹配成功，「^」必须能够匹配，但许多实现方式不能识别「^this|^that」。此时，用「^(this|that)」或者「^(?:this|that)」能够提高匹配的速度。

同样的优化措施还对「\A」有效，如果匹配多次进行，对「\G」也有效。

隐式锚点优化

能使用此种优化的引擎知道，如果正则表达式以「.*」或「.+」开头，而且没有全局性多选结构（global alternation），则可以认为此正则表达式的开头有一个看不见的「^」。这样就能使用上一节的"字符串起始/行锚点优化"，节省大量的时间。

更聪明的系统能够认识到，即使开头的「.*」或「.+」在括号内，也可以进行同样的优化，但是在遇到捕获括号时必须小心。例如，「(.+)X\1」期望匹配的是字符串在'X'两侧是相同的，添加「^」就不能匹配'1234X2345'（注5）。

字符串结束/行锚点优化

这种优化遇到末尾为「$」或者其他结束锚点（☞129）的正则表达式时，能够从字符串末尾倒数若干字符的位置开始尝试匹配。例如正则表达式「regex(es)?$」匹配只可能从字符串末尾倒数的第8个字符（注6）开始，所以传动装置能够跳到那个位置，略过目标字符串中许多可能的字符。

注5：有趣的是，Perl的这个"优化过度"的bug，在10年里都无人关注，最后由Perl开发人员Jeff Pinyan在2002年早期发现（并修正）。显然，「(.+)X\1」之类的表达式并不常见，否则这个bug就不会这么迟才被发现了。

注6：在这里，我说8个字符，而不是7个，因为在许多流派中，「$」能够匹配字符串末尾的换行符之前的位置（☞129）。

开头字符/字符组/子串识别优化

这是“预查必须字符/子字符串优化”的更一般的版本，这种优化使用同样的信息（正则表达式的任何匹配必须以特定字符或文字子字符串开头），容许传动装置进行快速子字符串检查，所以它能够在字符串中合适的位置应用正则表达式。例如，「this|that|other」只能从「[ot]」的位置开始匹配，所以传动装置预先检查字符串中的每个字符，只在可能匹配的位置进行应用，这样能节省大量的时间。能够预先检查的子串越长，“错误的开始位置”就越少。

内嵌文字字符串检查优化

这有点类似初始字符串识别优化，不过更加高级，它针对的是在匹配中固定位置出现的文字字符串。如果正则表达式是「\b(perl|java)\.regex\.info\b」，那么任何匹配中都要有‘.regex.info’，所以智能的传动装置能够使用高速的 Boyer-Moore 字符串检索算法寻找‘.regex.info’，然后往前数 4 个字符，开始实际应用正则表达式。

一般来说，这种优化只有在内嵌文字字符串与表达式起始位置的距离固定时才能进行。因此它不能用于「\b(vb|java)\.regex\.info\b」，这个表达式虽然包含文字字符串，但此字符串与匹配文本起始位置的距离是不确定的（2 个或 4 个字符）。这种优化同样也不能用于「\b(\w+)\.regex\.info\b」，因为「(\w+)」可能匹配任意数目的字符。

长度识别传动优化

此优化与 245 页的长度识别优化直接相关，如果当前位置距离字符串末尾的长度小于成功匹配所需最小长度，传动装置会停止匹配尝试。

优化正则表达式本身

Optimizations of the Regex Itself

文字字符串连接优化

也许最基本的优化就是，引擎可以把「abc」当作“一个元素”，而不是三个元素“「a」，然后是「b」，然后是「c」”。如果能够这样，整个部分就可以作为匹配迭代的一个单元，而不需要进行三次迭代。

化简量词优化

约束普通元素——例如文字字符或者字符组——的加号、星号之类的量词，通常要经过优

化，避免普通 NFA 引擎的大部分逐步处理开销（step-by-step overhead）。正则引擎内的主循环必须通用（general），能够处理引擎支持的所有结构。而在程序设计中，“通用”意味着“速度慢”，所以此种优化把「.*」之类的简单量词作为一个“整体”，正则引擎便不必按照通用的办法处理，而使用高速的，专门化的处理程序。这样，通用引擎就绕过（short-circuit）了这些结构。

举例来说，「.*」和「(?:.)*」在逻辑上是相等的，但是在进行此优化的系统中，「.*」实际上更快。举一些例子：在 java.util.regex 中，性能提升在 10%左右，但是在 Ruby 和.NET 中，大概是 2.5 倍。在 Python 中，大概是 50 倍。在 PHP/PCRE 中，大概是 150 倍。因为 Perl 实现了下一节介绍的优化措施，「.*」和「(?:.)*」的速度是一样的（请参考下一页的补充内容，了解如何解释这些数据）。

消除无必要括号

如果某种实现方式认为「(?:.)*」与「.*」是完全等价的，那么它就会用后者替换前者。

消除不需要的字符组

只包含单个字符的字符组有点儿多余，因为它要按照字符组来处理，而这么做完全没有必要。所以，聪明的实现方式会在内部把「[.]」转换为「\.」。

忽略优先量词之后的字符优化

忽略优先量词，例如「"(.*?)"」中的‘*?’，在处理时，引擎通常必须在量词作用的对象（点号）和「"」之后的字符之间切换。因为种种原因，忽略优先量词通常比匹配优先量词要慢，尤其是对上文中“化简量词优化”的匹配优先限定结构来说，更是如此。另一个原因是，如果忽略优先量词在捕获型括号之内，控制权就必须在括号内外切换，这样会带来额外的开销。

所以这种优化的原理是，如果文字字符跟在忽略优先量词之后，只要引擎没有触及那个文字字符，忽略优先量词可以作为普通的匹配优先量词来处理。所以，包含此优化的实现方式在这种情况下会切换到特殊的忽略优先量词，迅速检测目标文本中的文字字符，在遇到此文字字符之前，跳过常规的“忽略”状态。

此优化还有各种其他形式，例如预查一组字符，而不是特殊的一个字符（例如，检查「['"](.*?)["']」中的「['"]」，这有点类似前面介绍的开头字符识别优化）。

理解本章中的性能测试

本章的大多数性能测试给出的是某种语言的相对结果。例如，第 248 页我提到一种优化过结构能比另一种未优化的结构快 10%，至少在 Sun 的 Java regex package 中是这样。在.NET 中，两者之间的差异是 2.5 倍，在 PCRE 中，是 150 倍。在 Perl 中，两者是一样的（也就是，它们的速度是相同的）。

那么，你能够从中推导出不同语言之间的相对速度吗？显然不能。PCRE 中 150 倍的性能提升意味着优化执行的效果特别好，相对其他语言，或者意味着未优化的版本速度很慢。在大部分中，我几乎没有提到语言之间的计时问题，因为它们是语言开发人员争论的话题。

不过还是有一点值得一看，就是 Java 的 10%和 PCRE 的 150 倍背后的细节。看起来 PCRE 中未优化的「(?:.)*」的速度是 Java 的 1/11，不过优化的「.*」要快 13 倍。Java 和 Ruby 的优化版本速度相同，但是 Ruby 的未优化版本比 Java 的要慢 40%。Ruby 的未优化版本只比 Python 的未优化版本慢 10%，但是 Python 的优化版本比 Ruby 的优化版本快 20 倍。

所有这些都比 Perl 的要慢。Perl 的优化和未优化版本都比 Python 最快的版本快 10%。请注意，每种语言都有自己的强项，这些数字只是针对某个具体的测试情况而言的。

“过度”回溯检测

第 226 页的“实测”揭示的问题是，「(.+)*」之类的量词结合结构，能够制造指数级的回溯。避免这种情况的简单办法就是限定回溯的次数，在“超限”时停止匹配。在某些实际情况中这非常有用，但是它也为正则表达式能够应用的文本人为设置了限制。

例如，如果上限是 10 000 次回溯，「.*?」就不能匹配长于 10 000 的字符，因为每个匹配的字符都对应一次回溯。这种情况并不罕见，尤其在处理 Web 页时更是如此，所以这种限制非常糟糕。

出于不同的原因，某些实现方式限制了回溯堆栈的大小（也就是同时能够保存的状态的上限）。例如，Python 的上限是 10000。就像回溯上限一样，这也会限制正则表达式所能处理的文本的长度。

因为存在这个问题，本书中的某些性能测试构建起来非常困难。为了获得最准确的结果，性能测试中计时部分应该尽可能多地完成正则表达式的匹配，所以我创建了极长的字符串，比较例如「"(.*)"」，「"(.)*"」，「"(.)*?"」和「"([^"])*?"」的执行时间。为了保证结果有意义，我必须限制字符串的长度，以避免回溯计数或者堆栈大小的限制。你可以在 239 页看到这个例子。

避免指数级（也就是超线性 super-linear）匹配

避免无休止的指数级匹配的更好办法是，在匹配尝试进入超线性状态时进行检测。这样就能做些额外的工作，来记录每个量词对应的子表达式尝试匹配的位置，绕过重复尝试。

实际上，超线性匹配发生时是很容易检测出来的。单个量词“迭代”（循环）的次数不应该比目标字符串的字符数量更多。否则肯定发生了指数级匹配。如果根据这个线索发现匹配已经无法终止，检测和消除冗余的匹配是更复杂的问题，但是因为多选分支匹配次数太多，这么做或许值得。

检测超线性匹配并迅速报告匹配失败的副作用（side effect）之一就是，真正缺乏效率的正则表达式并不会体现出效率的低下。即使使用这种优化，避免了指数级匹配，所花的时间也远远高于真正需要的时间，但是不会慢到很容易被用户发现（不像是等到太阳落山一样漫长，而可能是多消耗 1/100s，对我们来说这很快，但对计算机来说很漫长）。

当然，总的来看可能还是利大于弊。许多人不关心正则表达式的效率——它们对正则表达式怀着一种恐惧心理，只希望能完成任务，而不关心如何完成。（你可能没见过这种情况，但是我希望这本书能够加强你的信心，就像标题说的那样，精通正则表达式）。

使用占有优先量词削减状态

由正常量词约束的对象匹配之后，会保留若干“在此处不进行匹配”的状态（量词每一轮迭代创建一个状态）。占有优先量词（☞142）则不会保留这些状态。具体做法有两种，一

种是在量词全部尝试完成之后抛弃所有备用状态，效率更高的办法是每一轮迭代时抛弃上一轮的备用状态（匹配时总需要保存一个状态，这样在量词无法继续匹配的时候引擎还能继续运转）。

在迭代中即时抛弃状态的做法效率更高，因为所占的内存更少。应用「.*」会在匹配每个字符时创造一个状态，如果字符串很长，会占用大量的内存。

自动"占有优先转换"

在第4章中（☞171），我们用「^\w+:」来匹配'Subject'。「\w+」匹配到字符串末尾时，最后的冒号无法匹配，所以回溯机制会强迫「\w+」逐个交还字符，在每个位置对「:」进行徒劳的尝试。在这个例子中，如果使用固化分组「^(?>\w+):」或者占有优先量词「^\w++:」，能够避免无谓的劳动。

聪明的实现方式应该能自动做到这一点。编译正则表达式时，引擎会检查**量词之后的元素**能匹配的字符是否与**量词作用元素**匹配的字符重叠，如果不重叠，量词就应该自动转变为占有优先的形式。

就我所知，目前还没有系统采用这种优化，在这里列出来，是鼓励开发人员考虑这个问题，因为我坚信它能带来实质性的收益。

量词等价转换

有人习惯用「\d\d\d\d」，也有人则习惯使用量词「\d{4}」。两者的效率有差别吗？对NFA来说，答案几乎是肯定的，但工具不同，结果也不同。如果对量词做了优化，则「\d{4}」会更快一些，除非未使用量词的正则表达式能够进行更多的优化。听起来有点迷惑？但事实确实如此。

我的测试结果表明，Perl、Python、PHP/PCRE和.NET中，「\d{4}」大概要快20%。但是，如果使用Ruby和Sun的Java regex package，「\d\d\d\d」则要快上好几倍。所以，看起来量词在某些工具中要更好一些，在另一些工具中则要差一些。不过，情况远非如此简单。

比较「====」和「={4}」。这个例子与上面的截然不同，因为此时重复的是确定的文字字符，而直接使用「====」引擎更容易将其识别为一个文字字符串。如果是，支持的高效的开头字符/

字符组/子串识别优化（☞247）就可以派上用场。对 Python 和 Sun 的 Java regex package 来说，情况正是如此，「====」比「={4}」快上 100 倍。

Perl、Ruby 和.NET 的优化手段更高级，它们不会区分「====」和「={4}」，结果，两者是一样快的（而且都比「\d\d\d\d」和「\d{4}」的例子快成百上千倍）。

需求识别

另一种简单的优化措施是，引擎会预先取消它认为对匹配结果没有价值的（例如，在不必捕获文本的地方使用了捕获型括号）工作。识别能力在很大程度上依赖于编程语言，不过这种优化实现起来也可以很容易，如果在匹配时能够指定某些选项，就能禁止某些代价高昂的特性。

Tcl 就能够进行这种优化。除非用户明确要求，否则它的捕获型括号并不会真正捕获文本。而.NET 的正则表达式提供了一个选项，容许程序员指定捕获型括号是否需要捕获。

提高表达式速度的诀窍

Techniques for Faster Expressions

之前的数页介绍了我见过的传统型 NFA 引擎使用的各种优化。没有任何程序同时具备所有这些优化，而且无论你爱用的程序目前支持哪些，情况也会随时间而改变。但是，理解可能进行的各种优化，我们就能写出效率更高的表达式。如果你还理解传统型 NFA 的工作原理，把这些知识结合起来，就可以从三方面获益：

- **编写适于优化的正则表达式** 编写适应已知（或者未来会支持的）优化措施的表达式。举例来说，「xx*」比「x+」能适用的优化措施更多，例如检查目标字符串中必须出现的字符（☞245），或者开头字符识别（☞247）。

- **模拟优化** 有时候我们知道所用的程序没有特殊的优化措施，但是通过手工模拟，我们还是能节省大量的时间。比如在「this|that」之前添加「(?=t)」，这样即使系统无法预知任何匹配结果必须以‘t’开头，我们还是能模拟开头字符识别（☞247），下文还会深入讨论这个例子。

- **主导引擎的匹配** 使用关于传统型 NFA 引擎工作原理的知识，能够主导引擎更快地匹配。拿「this|that」来说。每个多选分支都以「th」开头，如果第一个多选分支不能匹配「th」，第二个显然也不行，所以不必白费工夫。因此，我们可以使用「th(?:is|at)」。这样，「th」就只要检查一遍，只由在确实需要的时候才会用到代价相对高昂的多选结构功能。而且，「th(?:is|at)」开头的纯文字字符就是「th」，所以存在进行其他优化的可能。

重要的是认识到，效率和优化有时候处理起来比较麻烦。在阅读本节其他内容的时候，请不要忘记下面几点：

- 进行看来确实有帮助的改动，有时反而事与愿违，因为这样可能禁止了你所不知道的，已经生效的其他优化。
- 添加一些内容模拟你知道的不存在的优化措施，可能出现的情况是，处理那些添加内容的时间多于节省下来的时间。
- 添加一些内容模拟一个目前未提供的优化，如果将来升级以后的软件支持此优化，反而会影响或者重复真正的优化。
- 同样，控制表达式尝试触发某种当前可用的优化，将来某些软件升级之后可能无法进行某些更高级的优化。
- 为提高效率修改表达式，可能导致表达式难以理解和维护。
- 具体的修改带来的好处（或是坏处）的程度，基本上取决于表达式应用的数据。对某类数据来说有益的修改，可能对另一类数据来说是有害的。

我来举个极端点的例子：你在 Perl 脚本中找到「(000|999)$」，并决定把这些捕获型括号替换为非捕获型括号。你觉得这样速度更快，因为不再需要捕获文本的开销。但是奇怪的是，这个微小而且看起来有益的改动反而会把表达式的速度**降低许多个数量级**（比之前慢几千倍）。怎么会这样呢？原来在这里有若干因素共同作用，使用非捕获型括号时，字符串结束/行锚点优化（☞246）会被关闭。我不希望劝阻读者在 Perl 中使用捕获型括号——绝大多数情况下，使用他们都是有益的，但是在某些情况下，会带来灾难性的后果。

所以，检测并性能测试你期望实际应用的同类型的数据，有助于判断改动是否值得，但是，你仍然必须自己权衡众多因素。就说这么多，下面我开始讲解一些技巧，你能够用它们来挖掘引擎效率的最后一点潜能。

常识性优化

Common Sense Techniques

只需要依靠常识，就能进行一些极有成效的优化。

避免重新编译

编译和定义正则表达式的次数应该尽可能的少。在面向对象式处理中（☞95），用户能够精确控制这一点。举例来说，如果你希望在循环中应用正则表达式，就应该在循环外创建这个正则表达式对象，在循环中重复使用。

在函数式处理——例如GNU Emacs和Tcl——的情况下，应尽量保证循环中使用的正则表达式的数目少于工具所能缓存的上限（☞244）。

如果使用的是集成式处理，例如Perl，应尽量避免在循环内的正则表达式中使用变量插值，因为这样每次循环都需要重新生成正则表达式，即使值没有变化（不过，Perl提供了高效的办法来避免这个问题☞348）。

使用非捕获型括号

如果不需要引用括号内的文本，请使用非捕获型括号「(?:…)」（☞45）。这样不但能够节省捕获的时间，而且会减少回溯使用的状态的数量，从两方面提高速度。而且能够进行进一步的优化，例如消除无必要括号（☞248）。

不要滥用括号

在需要的时候使用括号，在其他时候使用括号会阻止某些优化措施。除非你需要知道「.*」匹配的最后一个字符，否则请不要使用「(.)*」。这似乎很显而易见，但是别忘了，本节的标题是“常识性优化”。

不要滥用字符组

这一点看起来也很显而易见，但是我经常在缺乏经验的程序员的表达式中看到「^.*[:]」。我不知道为什么他们会使用只包含单个字符的字符组——这样需要付出处理字符组的代价，

但并不需要用到字符组提供的多字符匹配功能。我认为，当一个字符是元字符时——例如「[.]」或者「[*]」，可能作者不知道通过转义把它们转换为「\.」和「*」。我经常在宽松排列模式（☞111）下见到它们与空白字符一起出现。

同样，读过本书第一版的Perl用户可能有时候写出「^[Ff][Rr][Oo][Mm]:」，而不是用不区分大小写的「^from:」。老版本的Perl在进行不区分大小写的匹配时效率很低，所以我推荐使用字符组。但现在这种做法已不再适用，因为不区分大小写匹配效率低下的问题在好几年前就修正了。

使用起始锚点

除非是极其罕见的情况，否则以「.*」开头的正则表达式都应该在最前面添加「^」或者「\A」（☞129）。如果这个正则表达式在某个字符串的开头不能匹配，那么显然在其他位置它也不能匹配。添加锚点（无论是手工添加，还是通过优化自动添加☞246）都能够配合开头字符/字符串/字串识别优化，节省大量不必要的工作。

将文字文本独立出来

Expose Literal Text

我们在本章见过的许多局部优化，主要是依靠正则引擎的能力来识别出匹配成功必须的一些文字文本。某些引擎在这一点上做得比其他引擎要好，所以这里提供了一些手动优化措施，有助于“暴露”文字文本，提高引擎识别的可能性，配合与文字文本有关的优化措施。

从量词中“提取”必须的元素

用「xx*」替代「x+」能够暴露匹配必须的‘x’。同样的道理，「-{5,7}」可以写作「------{0,2}」。

“提取”多选结构开头的必须元素

用「th(?:is|at)」替代「(?:this|that)」，就能暴露出必须的「th」。如果不同多选分支的结尾部分相同，我们也可以从右面“提取”，例如「(?:optim|standard)ization」。我们会在下一节中看到，如果提取出来的部分包括锚点，这么做就非常有价值。

将锚点独立出来

Expose Anchors

某些效果明显的内部优化措施是利用锚点（例如「^」、「$」和「\G」），把表达式“绑定”在目标字符串的某一端。使用这些优化时，某些引擎的效果不如其他引擎，但是有一些技巧能够提供帮助。

在表达式前面独立出^和\G

「^(?:abc|123)」和「^abc|^123」在逻辑上是等价的，但是许多正则引擎只会对第一个表达式使用开头字符/字符串/字串识别优化（☞246）。所以，第一种办法的效率要高得多。PCRE（还包括使用它的工具）中两者的效率是一样的，但是大多数其他 NFA 工具中第一个表达式的效率更高。

比较「(^abc)」和「^(abc)」我们能发现另一个区别。前者的设置并不很合适，锚点“藏”在捕获型括号内，在检测锚点之前，必须首先进入括号，在许多系统上，这样做的效率很低。某些系统（PCRE、Perl、.NET）中两者的效率相当，但是在其他系统中（Ruby 和 Sun 的 Java regex package）只会对后者进行优化。

Python 似乎不会进行锚点优化，所以这些技巧目前不适用于 Python。当然，本章介绍的大多数优化都不适用于 Tcl（☞243）。

在表达式末尾独立出$

此措施与上一节的优化思想非常类似，虽然「abc$|123$」和「(?:abc|123)$」在逻辑上是等价的，但优化的表现可能不同。目前，这种优化还只适用于 Perl，因为现在只有 Perl 提供了“字符串结束/行锚点优化”（☞246）。优化只对「(…|…)$」起作用，对「(…$|…$)」不起作用。

忽略优先还是匹配优先？具体情况具体分析

Lazy Versus Greedy: Be Specific

通常，使用忽略优先量词还是匹配优先量词取决于正则表达式的具体需求。举例来说，「^.*:」完全不同于「^.*?:」，因为前者匹配到最后的冒号，而后者匹配到第一个冒号。但是，如果目标数据中只包含一个分号，两个表达式就没有区别了（匹配到唯一的分号为止），所以选择速度更快的表达式可能更合适。

不过并不是任何时候优劣都如此分明，大的原则是，如果目标字符串很长，而你认为分号

会比较接近字符串的开头，就使用忽略优先量词，这样引擎能更快地找到分号。如果你认为分号在接近字符串末尾的位置，就使用匹配优先量词。如果数据是随机的，又不知道分号究竟会靠近哪一头，就使用匹配优先的量词，因为它们的优化一般来说都要比其他量词要好，尤其是表达式的后面部分禁止进行**“忽略优先量词之后的字符优化”**（☞248）时，更是如此。

如果待匹配的字符串很短，差别就不那么明显了。这时候，两个正则表达式的速度都很快，不过如果你很在乎那一点点速度差别，就对典型数据做个性能测试吧。

一个与此有关的问题是，在忽略优先量词和排除型字符组之间（「^.*?:」与「^[^:]*:」），应该如何选择？答案还是取决于所使用的编程语言和应用的数据，但是对大多数引擎来说，排除型字符组的效率比忽略优先量词高的多。Perl是个例外，因为它能对忽略优先量词之后的字符进行优化。

拆分正则表达式

Split Into Multiple Regular Expressions

有时候，应用多个小正则表达式的速度比一个大正则表达式要快得多。举个极端的例子，如果你希望检查一个长字符串中是否包含月份的名字，依次检查「January」、「February」、「March」之类的速度，要比「January|February|March|…」快得多。因为对后者来说，不存在匹配成功必须的文字内容，所以不能进行“内嵌文字字符串检查优化”（☞247）。“大而全”的正则表达式必须在目标文本中的每个位置测试所有的自表达式，速度相当慢。

撰写本章时，我遇到了一个有趣的情况。用Perl写一个数据处理模块时，我意识到客户端程序有个bug，导致它发送奇怪的数据，类似‘HASH(0x80f60ac)’而不是真正的数据。所以，我打算修正这个模块，寻找怪异数据的来源，并报告错误。我使用的正则表达式相当直白：「\b(?:SCALAR|ARRAY|…|HASH)\(0x[0-9a-fA-F]+\)」。

在这里，效率是非常关键的。这个表达式的速度如何？Perl的调试模式（debugging mode）能够告诉你它对特定表达式使用的某些优化（☞361），所以我进行了检查。我希望启用了预查必须字符/字符串优化（☞245），因为足够先进的引擎应该能够明白‘(0x’是任何匹配所必须的。因为这个正则表达式所应用的数据几乎不包含‘(0x’，我相信预查能够节省许多时间。不幸的是，Perl没有这样做，所以程序必须在每个目标字符串的每个字符那里测试整个正则表达式的众多多选分支。速度达不到我的要求。

因为正在研究和撰写与优化有关的内容，所以我冥思苦想，这个表达式应该怎样写才能得到优化。一个办法是以复杂的方式「\(0x(?<=(?:SCALAR|…|HASH)\(0x)[0-9a-fA-F]+\)」。这样，一旦「\(0x」匹配之后，肯定型顺序环视（下画线标注部分）就能确保之前匹配的是需要的文本，再检查之后的文本是否符合期望。费这番周折的原因在于，让正则表达式获得必须出现的文字文本「\(0x」，这样就能进行多种优化。尤其是，我希望进行预查必须字符串优化，以及开头字符/字符组/子串识别优化（☞247）。我确定这样速度会很快，但是Perl不支持长度不定的逆序环视（☞133），所以我得想办法来绕开它。

不过我发觉，如果Perl不会自动预查「\(0x」，我可以自己动手：

```
if ($data =~ m/\(0x/
    and
    $data =~ m/(?:SCALAR|ARRAY|…|HASH)\(0x[0-9a-fA-F]+\)/)
{
    #  错误数据，报警
}
```

「\(0x」的检查事实上会过滤大部分文本，相对较慢的完整正则表达式只对有可能匹配的行进行检测，这样就在效率（非常高）和可读性（非常高）之间达到了完美的平衡（注7）。

模拟开头字符识别

Mimic Initial-Character Discrimination

如果你的实现方式没有进行开头字符识别优化（☞247），则可以亲自动手，在表达式的开头添加合适的环视结构（☞133）。在正则表达式的其他部分匹配之前，环视结构可以进行“预查”，选择合适的开始位置。

如果正则表达式为「Jan|Feb|…|Dec」，对应的就是「(?=[JFMASOND])(?:Jan|Feb|…|Dec)」。开头的「[JFMASOND]」代表了英文中月份单词可能的开始字母。不过这种技巧并不是所有情况下都适用的，因为，环视结构的开销可能大于节省的时间。在这个例子中，因为多数多选分支都可能匹配失败，预查对我测试的大多数系统（Java、Perl、Python、Ruby、.NET）都是有用的，因为它们都不能自动从「Jan|Feb|…|Dec」得到开头的「[JFMASOND]」。

注7：你可以亲自去看看。提到的模块是（CPAN上的）DBIx::DWIW，它容许非常方便地访问MySQL数据库。由Jeremy Zawodny和我在Yahoo!开发。

在默认情况下，PHP/PCRE 并不会进行这种优化，但是如果使用 PCRE 的 pcre_study（也就是 PHP 中的模式修饰符 S，☞478）则可以。而 Tcl 显然能够完美地做到这一点（☞243）。

如果正则引擎能自动进行「[JFMASOND]」的检测，速度当然会比用户手工指定快得多。我们有没有办法让引擎自动进行检测呢？在许多系统上，我们可以用下面这个复杂得要命的表达式：

「[JFMASOND](?:(?<=J)an|(?<=F)eb|…|(?<=D)ec)」

我可不指望你能看一眼就明白这个表达式，但是花点时间仔细琢磨倒是很值得的。表达式开头的字符组可以被大多数系统的**开头字符识别**优化所利用，这样传动装置就能够高效地预查「[JFMASOND]」。如果目标字符串不包含匹配字符，结果会比原来的「Jan|…|Dec」或是手工添加环视功能的表达式要快。但是，如果目标字符串中包含许多字符组能够匹配的字符，那么额外的逆序环视可能反而会减慢匹配的速度。最主要的问题是，这个正则表达式显然很难看懂。但是，这个例子倒是很有意思，也很启发人。

不要在 Tcl 中这么做

上面的优化例子其实降低了表达式的可读性。243 页的补充内容提到，不同形式的正则表达式在 Tcl 中的表现是相同的，所以在 Tcl 中，大多数优化并没有意义。不过，有个例子中优化确实**有影响**。根据我的测试，手动添加「(?=[JFMASOND])」会把速度降低到原来的 1/100。

不要在 PHP 中这么做

之前我们已经提到过，不应该在 PHP 中进行优化，因为我们能够使用 PHP 的“study”功能——模式修饰符 S——来启用优化。详见第 10 章第 478 页。

使用固化分组和占有优先量词

Use Atomic Grouping and Possessive Quantifiers

在许多情况下，固化分组（☞139）和占有优先量词（☞142）能够极大地提高匹配速度，而且它们不会改变匹配结果。举例来说，如果「^[^:]+:」中的冒号第一次尝试时无法匹配，那么任何回溯其实都是没有意义的，因为根据定义，回溯“交还”的任何字符都不可能是冒号。使用固化分组「^(?>[^:]+):」或者占有优先量词「^[^:]++:」就能够直接抛弃备用状态，

或者根本不会创造多少备用状态。因为引擎没有内容状态可以回溯，就不会进行不必要的回溯（第251页的补充内容说明，足够聪明的引擎能够自动进行这种优化）。

不过，我必须强调，这两种结构运用不当，就会在不经意间改变匹配结果，所以必须极为小心。如果不用「^.*:」而用「^(?>.*):」，结果必然会失败。整行文本都会被「.*」匹配，后面的「:」就无法匹配任何字符。固化分组阻止最后的「:」匹配必须进行的回溯，所以匹配必定失败。

主导引擎的匹配

Lead the Engine to a Match

提高正则表达式匹配效率的另一个办法是尽可能准确地设置匹配过程中的"控制权"。我们曾经看过的用「th(?:is|at)」取代「this|that」的例子。在后一个表达式中，多选结构获得最高级别的控制权，而在前一个表达式中，相对代价更高昂的多选结构只有在「th」匹配之后才获得控制权。

下一节"消除循环"是这种技巧的高级形式，此处再介绍些简单的技巧。

将最可能匹配的多选分支放在前头

在本书中我们看到，许多时候多选分支的摆放顺序非常重要（☞28、176、189、216）。在这些情况下，摆放顺序比优化更重要，但相反，如果顺序与匹配正确无关，就应该把最可能匹配的多选分支放在首位。

举例来说，在匹配主机名的正则表达式（☞205）中，有人可能习惯把后缀按照字母顺序排序，例如「(?:aero|biz|com|coop|…)」。不过，某些排在前头的后缀应用并不广泛，匹配极有可能失败，为什么要把他们排在前头呢？如果按照分布数量的排序：「(?:com|edu|org|net|…)」，更有可能获得更迅速更常见的匹配。

当然，这只有对传统型NFA引擎才适用，而且只有存在匹配的时候才适用。如果使用POSIX NFA，或是不存在匹配，此时所有的多选分支都必须检测，所以顺序是无关紧要的。

将结尾部分分散到多选结构内

接下来我们比较「(?:com|edu|…|[a-z][a-z])\b」和「com\b|edu\b|…\b|[a-z][a-z]\b」。在后一个表达式中，多选结构之后的「\b」被分散到每个多选分支。可能的收益就是，它可能容许一个多选分支能够匹配，但多选分支之后的「\b」可能导致这个匹配不成功，把「\b」加到多选结构之内，匹配失败的更快。这样不需要退出多选结构就能发现失败。

要说明这个技巧的价值，这可能还不是最好的办法，因为它只是适用于一种特殊情形，即多选分支可能能够匹配，但是之后紧接的字符可能令匹配失败。在本章中我们会看到一个更合适的例子——请参考 280 页关于$OTHER*的讨论。

这个优化是有风险的。切记，使用这个功能时要小心，不要因此阻止了本来可以进行的其他优化。举例来说，如果“分散的”子表达式是文字文本，那么把「(?:this|that):」更换为「this:|that:」就违背了“将文字文本独立出来”（☞255）中的某些思想。各种优化都是平等的，所以在优化时请务必小心，不要因小失大。

在能够进行独立结尾锚点（☞256）的系统上把正则表达式末尾的「$」分散，也会遇到这种问题。在这些系统上，「(?:com|edu|…)$」比「com$|edu$|…$」快得多（我测试了各种系统，只有 Perl 使用了这种优化）。

消除循环

Unrolling the Loop

无论系统本身支持怎样的优化，最重要的收益或许还是来自于对引擎基本工作原理的理解，和编写能够配合引擎工作的表达式。所以，既然我们已经考察了繁琐的细节，不妨登堂入室，学习我说的“消除循环”的技巧。对某些常用的表达式来说，它的加速效果很明显。举例来说，用它改造本章开头（☞226）会进行无休止匹配的表达式，能够在有限的时间内报告匹配失败，而如果能够匹配，速度也会更快。

此处说的“循环”采用的是「(*this*|*that*|…)*」之类问题中星号代表的意义。之前的无休止匹配「"(\\.|[^\\"]+)*"」其实就属于此类。如果无法匹配，这个表达式需要近乎无限的时间进行尝试，所以必须改进。

此技巧有两种不同的实现途径：

1. 我们可以检查，在各种典型匹配中，「(\\.|[^\\"]+)*」中的哪个部分真正匹配成功了，这样就能留下子表达式的痕迹。再根据刚刚发现的模式，重新构建高效的表达式。这个（或许联系不那么紧密的）概念模型就是一个大球，它表示表达式「(…)*」，球在某些文本上滚动。「(…)」内的元素总是能够匹配某些文本，这样就留下了痕迹，就好像是把脏兮兮的雪球在地毯上滚过去一样。

2. 另一个办法是，从更高的层面考察我们期望用于匹配的结构。然后根据我们认为的常见情形，对可能出现的目标字符串做出非正式假设。从这个角度出发构建有效的表达式。

无论采取哪种办法，得到的表达式都是一样的。我首先讲解第一种思路，然后介绍如何通过第二种思路取得同样的结果。

为了保证例子容易看懂，并尽可能广泛地使用，我会使用「(…)」来代替所有括号，如果程序支持非捕获型括号「(?:…)」能够支持，使用它们能提高效率。然后会讲解固化分组（☞139）和占有优先量词（☞142）的使用。

方法1：依据经验构建正则表达式

Method 1: Building a Regex From Past Experiences

在分析「"(\\.|[^\\"]+)*"」时，用若干具体的字符串来检验全局匹配的具体情况是很自然的做法。举例来说，如果目标字符串是‘"hi"’，那么使用的自表达式就是「"[^\\"]+"」。这说明，全局匹配使用了最开始的「"」，然后是多选分支「[^\\"]+」，然后是末尾的「"」。

如果目标字符串是：

```
"he said \"hi there\" and left"
```

对应的表达式就是「"[^\\"]+\\.[^\\"]+\\.[^\\"]+"」。在这个例子里以及表6-2中，我标记了对应的正则表达式，让模式更显眼。如果我们能对每个输入字符串构造特定的表达式当然很好。这不可能，但我们能够找出一些常用的模式，构造效率更高，又不失通用性的正则表达式。

现在来看表6-2前面的四个例子。下画线标注的部分表示“一个转义元素，然后是更多的正常字符”。这就是关键：在每种情况下，开头是引号，然后是「[^\\"]+」，然后是若干个「\\.[^\\"]+」。综合起来就得到「[^\\"]+(\\.[^\\"]+)*」。这个特殊的例子说明，通用模式可以用来构建许多有用的表达式。

表 6-2：消除循环的具体情况

目标字符串	对应的表达式
"hi there"	"[^\\"]+"
"just one \" here"	"[^\\"]+\\.[^\\"]+"
"some \"quoted\" things"	"[^\\"]+\\.[^\\"]+\\.[^\\"]+"
"with \"a\" and \"b\"."	"[^\\"]+\\.[^\\"]+\\.[^\\"]+\\.[^\\"]+\\.[^\\"]+"
"\"ok\"\n"	"\\.[^\\"]+\\.\\."
"empty \"\" quote"	"[^\\"]+\\.\\.[^\\"]+"

构造通用的“消除循环”解法

在匹配双引号字符串时，引号自身和转义斜线是“特殊”的——因为引号能够表示字符串的结尾，反斜线表示它之后的字符不会终结整个字符串。在其他情况下，「[^\\"]」就是普通的点号。考察它们如何组合为「[^\\"]+(\\.[^\\"]+)*」，首先它符合通用的模式「*normal*+(*special normal*+)*」。

再添加两端的引号，就得到「"[^\\"]+(\\.[^\\"]+)*"」。不幸的是，表 6-2 中下面两个例子无法由这个表达式匹配。症结在于，目前这个正则表达式的两个「[^\\"]+」**要求**字符串以一个普通字符开始，然后是任何数目的特殊字符。从这个例子中我们可以看到，它并不能应付所有情况——字符串可能以转义元素开头和结尾，一行中间也可能包含两个转义元素。

我们可以尝试把两个加号改成星号：「"[^\\"]*(\\.[^\\"]*)*"」。这会达到我们期望的结果吗？更重要的是，它是否会产生负面影响？

就期望的结果来说，很容易看到所有的例子都能匹配。事实上，即使是"\"\"\""这样的字符串都能匹配。这当然很不错。不过，我们还需要确认，这样重大的改变，是否会导致预料之外的结果。格式不对的引号字符串能否匹配呢？格式正确的引号字符串是否可能无法匹配呢？效率又如何呢？

仔细看看「"[^\\"]*(\\.[^\\"]*)*"」。开头的「"[^\\"]*」只会应用一次，这没有问题：它匹配开头必须出现的引号，以及之后的任何普通字符。这没有问题。接下来的「(\\.[^\\"]*)*」是由星号限定的，所以不匹配任何字符也能够成功。也就是说，去掉这一部分仍然会得到一个合法的表达式。这样我们就得到「"[^\\"]*"」，这显然没有问题——它代表了常见的，也就是没有转义元素的情形。

另一方面，如果「(\\.[^\\"]*)*」部分匹配了一次，其实就等价于「"[^\\"]*\\.[^\\"]*"」。

即使结尾的「[^\\"]*」没有匹配任何文本（其实就是「"[^\\"]*\\."」），也没有问题。照这样分析下去（如果没记错的话，这就是高中代数中的“照此类推（by induction）”），我们发现，这样的改动其实是没有任何问题的。

所以，我们最后得到的，用来匹配包括转义引号的引号字符串的正则表达式就是：

「"[^\\"]*(\\.[^\\"]*)*"」

真正的“消除循环”解法

The Real "Unrolling-the-Loop" Pattern

综合起来，匹配包括转义引号的双引号字符串正则表达式就是「"[^\\"]*(\\.[^\\"]*)*"」。这和原来的表达式能够匹配的结果是完全一致的。但是循环消除之后，表达式能够在有限的时间内结束匹配。不但效率高得多，并且避免了无休止匹配。

消除循环常用的解法是：

「*opening normal* (special normal*)* closing*」

避免无休止匹配

避免「"[^\\"]*(\\.[^\\"]*)*"」中的无休止匹配，有三点很重要：

1）special 部分和 normal 部分匹配的开头不能重合。

special 和 *normal* 部分的子表达式不能从同一位置开始匹配。在上例中，*normal* 部分是「[^\\"]」，*special* 部分是「\\.」，显然它们不能匹配同样的字符，因为后者要求以反斜线开头，而前者不容许出现开头的反斜线。

另一方面，「\\.」和「[^"]」都能够从‘"Hello\n"’开始匹配，所以它们不符合这种解法。如果二者能够从字符串中的同一位置开始匹配，就无法确定该使用哪一个，这种不确定就会造成无休止匹配。‘makudonarudo’的例子说明了这一点（☞227）。如果无法匹配（或是POSIX NFA 引擎在任何情况下的匹配），就必须尝试所有的可能性。这非常糟糕，因为改进这个表达式的首要原因就是为了避免这种情况。

如果我们确信，*special* 和 *normal* 部分不能匹配同样的字符，就可以将 *special* 部分用作检查点，消除 *normal* 部分在「(…)*」的各轮迭代时匹配同样文本造成的不确定性。如果我们确信 *special* 部分和 *normal* 部分永远不会匹配同样的文本，则特定目标字符串的匹配中存在唯一的 *special* 部分和 *normal* 部分的“组合序列”。检查这个序列比检查成千上万种可能要快得多，于是避免了无休止匹配。

2）normal 部分必须匹配至少一个字符

第二点是，如果 *normal* 部分需要匹配字符才能成功，则它必须匹配至少一个字符。如果我们能够匹配成功，而不占用任何字符，那么下面的字符仍然必须由「(*special normal**)*」的不同迭代来匹配，这样我们就回到了原来的(…*)*的问题。

选择「(\\.)*」作为 *special* 部分就违背了这条规定。「"[^\\"]*((\\.)*[^\\."]*)*"」注定是个糟糕的表达式，如果用它来匹配‘"Tubby’（会失败），在得到匹配失败的结论之前，引擎必须尝试若干个「[^\\"]*」匹配‘Tubby’的每一种可能。因为 *special* 部分可以不匹配任何字符，所以它无法作为检查点。

3）special 部分必须是固化的

special 部分匹配的文本不能由该部分的多次迭代完成。如果需要匹配 Pascal 中可能出现的注释{…}和空白。能够匹配注释部分的正则表达式是「\{[^}]*\}」，所以整个正则表达式就是「(\{[^}]*\}|·+)*」（它永远不会终止）。或许，你会这样划分 *normal* 和 *special* 部分：

Special	normal
「·+」	「\{[^]}*\}」

使用我们学会的「*normal** (*special normal**)*」的解法，我们得到「(\{[^]}*\})」*(·+(\{[^}]*\})*)*」。现在来看这个字符串

```
{comment}···{another}··
```

匹配连续空格的可能是单个「·+」，或多个「·+」匹配（每个匹配一个空格），或是多个「·+」的组合（每个匹配不同数目的空格）。这很像之前的‘makudonarudo’的问题。

此问题的根源在于，*special* 部分既能够匹配很长的文本，也能通过(…)*匹配其中的部分文本。非确定性开了“多种方式匹配同样文本”的口子。

如果存在全局匹配，很可能「·+」只匹配一次，但是如果不存在全局匹配（例如把这个表达式作为另一个大的表达式的一部分），引擎就必须对每一段空格测试「(·+)*」所有的可能。这需要时间，但这对全局匹配无益。因为 *special* 部分应该作为检查点，而这里没有任何需要检查的非确定性。

解决的方法就是，保证 *special* 部分只能够匹配固定长度的空格。因为它必须匹配至少一个空格，但可能匹配更多，我们用「·」作为 *special* 部分，用「(…)*」来保证 *special* 的多重应用能匹配多个空格。

这个例子很适合讲解，但是实际应用起来，效率更高的办法可能是交换 *special* 和 *normal* 表达式：

「·*(\{[^]}*\}·*)*」

因为我估计Pascal程序的注释不会比空格少，而且对常见情况来说更有效的办法是用*normal*部分匹配常见的文本。

寻找通用套路

如果你真正掌握了这些规则（可能需要反复阅读和一些实践），你就能把这几条推广开来，作为指导规则，用来识别可能造成无休止匹配的正则表达式。如果有若干个量词存在于不同的层面，例如「(…*)*」，我们就必须小心对待，但是许多这样的表达式却是完全没有问题的。例如：

- 「(Re:·*)*」用来匹配任意数目的‘Re:’序列（可以用来清除邮件主题中的‘Subject:·Re:·Re:·Re:·hey’）。
- 「(·*\$[0-9]+)*」用来匹配美元金额，可能有空格作分隔。
- 「(.*\n)+」用来匹配一行或多行文本。（事实上，如果点号不能匹配换行符，而这个子表达式之后又有别的元素导致匹配失败，就会造成无休止匹配）。

这些表达式都没有问题，因为每一个都有检查点，也就不会产生“多种方式匹配同样文本”的问题。在第一个里面是「Re:」，第二个里面是「\$」，第三个（如果点号不能匹配换行符）是「\n」。

方法2：自顶向下的视角

Method 2: A Top-Down View

还记得吗？我说过，“消除循环”有两种办法。如果采用第二种办法，开始只匹配目标字符串中最常见的部分，然后增加对非常见情况的处理。让我们来看会造成无休止匹配的表达式「(\\.|[^\\"]+)*」期望匹配的文本以及它可能应用的场合。我认为，通常情况下引用字符串中的普通字符会比转义字符更多，所以「[^\\"]+」承担了大部分工作。「\\.」只需要用来处理偶然出现的转义字符。我们可以使用多选结构来应付两种情况，但糟糕的是，为了处理少部分（决不可能是大部分）转义字符，这样做会降低效率。

如果我们认为「[^\\"]+」能够匹配字符串中的绝大部分字符，就知道如果匹配停止，大概是遇到了闭引号或者是转义字符。如果是转义字符，后面容许出现更多的字符（无论是什么），然后开始「[^\\"]+」的新一轮匹配。每次[^\\"]+的匹配终止，我们都回到相同的处境：期望出现一个闭引号或者是另一个转义。

我们可以很自然地用一个表达式来表达它，得到与方法 1 同样的结果：

「"[^\\"]+(\\.[^\\"]+)*"」

我们知道，匹配每次进行到标记的位置时，应该出现一个反斜线或者闭双引号。如果反斜线能匹配，就匹配它，然后是被转义的字符，然后是更多的字符，直到下一次到达"闭引号或者反斜线"的位置。

和之前一样，最开始的非引用内容或是引号内的文本可能为空。我们可以把两个加号改成星号，这样就得到与 264 页相同的表达式。

方法 3：匹配主机名

Method 3: An Internet Hostname

上面介绍了两种**消除循环**的办法，不过我还愿意介绍另一种办法。在用正则表达式匹配如 www.yahoo.com 这样的主机名时，它令我震惊。主机名主要是用点号分隔的子域名的序列，准确地划定子域名的匹配规范很麻烦（☞203），为了保证清晰，我们使用「[a-z]+」来匹配子域名。

如果子域名是「[a-z]+」，我们希望得到点号分隔的子域名序列，首先要匹配第一个子域名。之后其他的子域名以点号开头。用正则表达式来表示，就是「[a-z]+(\.[a-z]+)*」。现在，如果我希望添加上前面出现的各种标记，就得到「[a-z]+(\.[a-z]+)*」，显然它看起来非常熟悉，对吗？

为了说明这种相似性，我们尝试把它对应到双引号字符串的例子。如果我们认为字符串是'"…"'之内的文本，包括 *normal* 部分「[^\\"]」，由 *special* 部分「\\.」分隔，就能套用消除循环的解法，得到「"[^\\"]+(\\.[^\\"]+)*"」中，也就是在讨论方法 1 中的某个地步。也就是说，从概念上讲，我们能够把由点号分隔的主机名的问题看成双引号字符串的问题，也就是"由转义元素分隔的非转义元素构成的序列"。这可能不那么直观，但是我们可以使用前面用过的套路。

二者既存在相似性，也存在区别。在方法 1 中，我们改变正则表达式是为了容许 *normal* 部分和 *special* 部分之后出现空白，但是这里不容许出现空白。所以虽然这个例子与之前的并非完全相同，但也属于同一类，同样可以用来证明消除循环的技巧的强大和便捷。

子域名的例子与之前的例子有两大区别：

- 域名的开始和结束没有分界符。
- 子域名的 *normal* 部分不可能为空（也就是说两个点号不能紧挨在一起，点号也不能出现在整个域名的开头或结尾）。对双引号字符串来说，*normal* 部分可以为空，即使按照我们的假设它们不太应该为空。所以我们需要把「[^\\"]+」改为「[^\\"]*」。而子域名的例子不能进行这种修改，因为 *special* 部分是分隔符，必须出现。

观察

Observations

回过头来看双引号字符串的例子，表达式「"[^\\"]*(\\.[^\\"]*)*"」有许多优点，也存在一些陷阱。

陷阱：

- **可读性** 这是最大的问题，原来的「"([^\\"]|\\.)*"」可能更容易一眼看懂，我们放弃了可读性来追求效率。
- **可维护性** 可维护性可能更复杂，因为任何改动都必须保持对两个[^\\"]相同。我们牺牲了可维护性来追求效率。

好处：

- **速度** 如果不能匹配，或是采用 POSIX NFA，这个正则表达式不会进入无休止匹配。因为进行了精心地调校，特定的文本只能以唯一的方式匹配，如果文本不能匹配，引擎会迅速发现它。
- **还是速度** 正则表达式“操作连续性（flow）”很好，这也是“流畅运转的正则表达式”（☞277）的主题。我对传统型 NFA 进行了检测，消除循环之后的表达式总是比之前使用多选结构的表达式要快得多。即使匹配能够成功，不会进入无休止匹配状态时，也是如此。

使用固化分组和占有优先量词

Using Atomic Grouping and Possessive Quantifiers

表达式「"(\\.|[^\\"]+)*"」之所以会进入无休止匹配的状态，问题在于，如果无法匹配，它会陷入徒劳的尝试。不过，如果存在匹配，就能很快结束，因为「[^\\"]+」能够匹配目标字符串中的大多数字符（也就是之前讨论过的 *normal* 部分）。因为「[…]+」通常会为速度优化（☞247），而且能够匹配大多数字符，外面的「(…)*」量词的开销因此大为减少。

所以，「"(\\.|[^\\"]+)*"」的问题就在于，不会在匹配时会陷入徒劳的尝试，在我们知道毫无用处的备用状态之中不断回溯。我们知道这些状态毫无价值，因为他们只是检查同样对象的不同排列（如果「abc」不能匹配‘foo’，那么「abc」或者「abc」（以及「abc」，「abc」或者无论什么形式的「abc」）都不能匹配。如果我们能抛弃这些状态，正则表达式就能迅速报告匹配失败）。

抛弃（或者是忽略）这些状态的方法有两种：固化分组（☞139）或者占有优先量词（☞142）。

在我们着手消除回溯以前，我希望交换多选分支的顺序，把「"(\\.|[^\\"]+)*"」变为「"([^\\"]+|\\.)*"」，这样匹配“普通”文本的元素就出现在第一位。前几章中我们已经数次提到过，如果两个或两个以上的多选分支能够在同一位置匹配，排列顺序的时候就要小心，因为顺序可能影响到匹配的结果。但对于本例来说，不同多选分支匹配的是不同的文本（某个多选分支在一处能够匹配，则其他多选分支在此处就不能匹配），从正确匹配的角度来看，顺序就是无关紧要的，所以我们可以根据清晰或效率的要求来选择顺序。

使用占有优先量词避免无休止匹配

会造成无休止匹配的表达式「"([^\\"]+|\\.)*"」有两个量词。我们可以把其中一个改为占有优先量词，或者两个都改。这两者有区别吗？因为大多数回溯的麻烦源自「[…]+」留下的状态，所以把它改成占有优先是我的第一想法。这样得到的表达式，即使找不到匹配，速度也很快。不过，把外面的「(…)*」改成占有优先会抛弃括号内的所有备选状态，其中包括「[…]+」和多选结构本身的备选状态，所以如果我要从中选取一个的话，我会选取后者。

但我们并非只能选择一个，因为我们可以把两个都改为占有优先量词。具体哪种办法最快，可能取决于占有优先量词的优化情况。现在，只有 Sun 的 Java regex package 支持这种表示法，所以我的测试只能在 Java 中进行，并且发现某些情形下其中一种组合更快。我原本期望，使用两个占有优先量词是最快的，所以这些结果让我相信，Sun 的优化还不够彻底。

使用固化分组避免无休止匹配

如果要对「"([^\\"]+|\\.)*"」使用固化分组，最容易想到的办法就是把普通括号改成固化分组括号：「"(?>[^\\"]+|\\.)*"」。不过我们必须知道，在抛弃状态的问题上，「(?>…|…)*」与占有优先的「(…|…)*+」是迥然不同的。

「(…|…)*+」在完成时不会留下任何状态，相反，「(?>…|…)*」只是消除多选结构每次迭代时保留的状态。星号是**独立于**固化分组的，所以不受影响，这个表达式仍然会保留“跳过本轮迭代”的备用状态。也就是说，回溯中的状态仍然不是确定的最终状态。我们希望同时消除外面量词的备用状态，所以要把外面的括号也改成固化分组。也就是说模拟占有优先「(…|…)*+」必须用到「(?>(…|…)*)」。

解决无休止匹配的问题时，「(…|…)*+」和「(?>…|…)*」都很有用，但是它们在抛弃状态的选择和时间上却是不同的（更多的差异，请参阅173页）。

简单的消除循环的例子

Short Unrolling Examples

现在我们大概了解了消除循环的思想，来看看书中曾经出现过的几个例子，想想该如何消除循环。

消除“多字符”引文中的循环

在第4章第167页，我们看到这个例子：

```
<B>                    # 匹配开头的<B>
(                      # 现在匹配尽可能多的…
    (?! </?B> )        # 如果不是<B>也不是</B> …
    .                  # …任何字符都没问题
)*                     # (匹配优先量词)
</B>                   # <ANNO>直到结束边界
```

normal 部分是「[^<]」，*special* 部分是「(?!</?B>)<」，下面是改进的版本：

```
<B>                        # 匹配开头的 <B>
    (?> [^<]*)             # 匹配任意数量的"normal"…
    (?>                    # 任意数量的…
        (?! < /? B> )      # 如果不是<B>也不是</B>
        <                  # 匹配"special"
        [^<]*              # 然后继续匹配任意数量的"normal"
    )*                     #
</B>                       # 最后匹配结尾的</B>
```

这里固化分组并不是必须的，但如果只能部分匹配，使用固化分组能够提高速度。

消除连续行匹配例子中的循环

连续行的例子出现在前一章的开头（☞186），当时使用的表达式是「^\w+=([^\n\\]|\\.)*」。看起来这很适合应用这种技巧：

```
^ \w+ =                         # 开头的文字和'='
# 现在读取（并且捕获）值...
(
    (?> [^\n\\]* )              # "normal"*
    (?> \\. [^\n\\]* )*         # ( "special" "normal"*)*
)
```

与上一个例子一样，固化分组不是必须的，但它能让引擎更快地报告匹配失败。

消除 CSV 正则表达式中的循环

第 5 章用了很长的篇幅讨论 CSV 的处理，最后得到第 216 页的代码：

```
(?:^|,)
(?:#  或者是匹配双引号字段（其中容许出现连在一起的成对双引号）...
      " #   (起始双引号)
       ( (?: [^"] | "" )* )
      " #   (结束双引号)
  |
   #  ... 或者是引号和逗号之外的文本...
        ( [^",]* )
)
```

当时的结论是，最好在开头添加「\G」，这样就能避免驱动过程带来的麻烦，并且效率也会提高。现在我们知道如何消除循环，就可以此技巧来看看如何应用这个例子。

用来匹配微软的 CSV 字符串的正则表达式是「(?:[^"]|"")*」，它看起来很不错。其实，这个表达式已经区分了 *normal* 和 *special* 部分：「[^"]」和「""」。下面我们把这个表达式写清楚，用原来的 Perl 代码说明消除循环的过程：

```
while ($line =~ m{
            \G(?:^|,)
            (?:
                #  或者匹配双引号字段（其中容许出现连在一起的成对双引号）?
                "  #  起始双引号
                  ( (?> [^"]* ) (?> "" [^"]* )*  )
                "  #  结束双引号
            #  ...或者是
            |
               #  ... 或者是引号和逗号之外的文本...
               ( [^",]* )
            )
        }gx)
    {
       if (defined $2) {
           $field = $2;
       } else {
           $field = $1;
           $field =~ s/""/"/g;
       }
    print "[$field]"; #  输出字段的值以供调试
    现在处理$field...
}
```

如其他的例子一样，固化分组不是必须的，但可以提高效率。

消除C语言注释匹配的循环

Unrolling C Comments

现在来看个匹配更复杂字符串时消除循环的例子。在C语言中，注释以/*开头，*/结尾，可以有多行，但不能嵌套（C++、Java和C#也容许这种形式的注释）。匹配此类注释的正则表达式在许多情况下都有用，例如构建去掉注释的过滤程序。写这个程序时我首先想到的就是消除循环，而这个技巧现在已经成为我的正则表达式宝库中的重要装备。

真的需要消除吗

我在20世纪90年代早期就开始开发本节讨论的这个正则表达式。在那之前，人们认为用正则表达式匹配C语言的注释即使不是不可能，也是很困难的事情，所以一些可行的办法由我开发出来之后，就成为匹配C语言注释的标准方法。不过，在Perl引入忽略优先量词之后，出现了简单得多的办法：使用能匹配所有字符的点号「/*.*?*/」。

在我写程序的时候忽略优先量词还没有出现，如果当时有这种现成的特性，就不用费这么多周折了。不过，我的解决办法仍然是有效的，因为即使在首次支持忽略优先量词的那一版Perl中，使用消除循环技巧的程序仍然要比使用忽略优先量词的快得多（我做了许多种测试，有时快50%，也有时快360%）。

不过，Perl现在综合了各种优化措施，形势就颠倒过来，忽略优先量词的程序要快上50%到550%。所以我现在使用「/*.*?*/」来匹配C语言的注释。

这是否意味着，现在匹配C语言注视用不着消除循环的技巧了？如果引擎不支持忽略优先量词，消除循环的价值就能体现出来。也不是所有的正则引擎都能综合各种优化：在我测试的其他任何语言中，消除了循环的程序都要更快——最快的时候速度相差60倍！消除循环的技巧确实很有用，所以下文讲解如何用它来匹配C语言注释。

因为匹配C语言注释时不存在双引号字符串中转义字符\"的问题，可能有人觉得事情会比较简单，但问题其实更复杂。因为这里的“结束双引号”*/不止一个字符。直接用「/*[^*]**/」可能看起来没问题，但不能匹配 `/**some comment here**/`，因为其中还有‘*’，而这是必须匹配的，所以我们需要另外的办法。

换更清晰的表示方法

你可能觉得「/*[^*]**/」难以阅读，即使本书的体例已经尽量做到容易看懂。但不幸的是，注释部分的边界符‘*’本身就是正则表达式的元字符，所以得使用反斜线转义，结果正则表达式看起来让人头疼。为了看得更清楚，我们在这个例子中使用/x…x/，而不是/*…*/。经过这个细微的改动，「/*[^*]**/」变成了更容易看懂的「/x[^x]*x/」。这个表达式会随着我们的讲解变得越来越复杂，到时你会发现这个改动的价值。

直接的办法

在第5章（☞196），我给出了匹配分隔符之内文本的公式：

1. 匹配起始分隔符；

2. 匹配正文：匹配“除结束分隔符之外的任何字符”；

3. 匹配结束分隔符。

现在我们的程序以/x和x/作为开始和结束分隔符，它似乎很符合这个模式。难处在于匹配“除结束分隔符之外的任何字符”。如果结束分隔符是单个字符，我们可以用排除型字符组。但字符组不能用来进行多字符匹配，不过如果能使用否定型顺序环视，我们就能使用「(?:(?!**x/**).)*」。这就是「**除结束分隔符之外的任何字符**」。

于是我们得到「/x(?:(?!**x/**).)*x/」。它没有问题，但速度很慢（在我做的一些测试中，速度要比下面的表达式慢几百倍）。这个思路很有用，但缺乏实用性，因为几乎所有支持顺序环视的流派都支持忽略优先量词，所以效率并不是问题，你完全可以用「/x.*?x/」。

那么，顺着这种分三步走的思路，是否有其他办法匹配第一个x/之前的文本？能想到的办法有两个。之一是把x作为开始分隔符和结束分隔符，也就是说，匹配除x之外的任何字符，以及之后字符不为斜线的x。这样，“除结束分隔符之外的任何字符”就成了：

- 除x之外的任何字符：「[^x]」。
- 之后字符不是斜线的x：「x[^/]」。

这样得到「([^x]|x[^/])*」来匹配主体文本，「/x([^x]|x[^/])*x/」来匹配整个注释。我们会发现这条路行不通。

另一种办法是，把紧跟在 x 之后斜线当作结束分隔符。这样“除结束分隔符之外的任何字符”就成了：

- 除斜线外的任何字符：「[^/]」。
- 紧跟在 x 之后的斜线：「[^x]/」。

于是用「([^/]|[^x]/)*」匹配主体文本，「/x([^/]|[^x]/)*x/」匹配整个注释。

不幸的是，这同样是死路。

如果用「/x([^x]|x[^/])*x/」来匹配‘/xx·foo·xx/’，在‘foo·’之后，第一个 x 由「x[^/]」匹配，这当然没有问题。但是之后，「x[^/]」匹配 xx/，而这个 x 应该是标记注释的结束。于是继续下一轮迭代，「[^x]」匹配斜线，结果会匹配 x/之后的文本。

「/x([^/]|[^x]/)*x/」也不能匹配‘**/x**/·foo·/**x/**’（整个注释都应该匹配）。如果注释结尾后紧跟斜线，表达式匹配的内容会超过注释的结束分隔符（这也是其他解法的问题）。而在本例中，回溯可能有些令人迷惑，所以读者最好弄明白「/x([^/]|[^x]/)*x/」为什么能匹配

```
Years = days /x divide x//365, /x assume non-leap year x/
```

（可以在空余时间好好想想这个问题。）

怎么办

让我们来修正这些表达式。在第一种情况下，因为疏忽，「x[^/]」匹配了结尾的…xx/。如果我们用「/x([^x]|x+[^/])*x/」。我们认为，添加加号之后，「x+[^/]」匹配以非斜线字符结尾的一连串 x。确实它能够这样匹配，但因为回溯“斜线之外的任意字符”仍然可以是 x。首先，匹配优先的「x+」匹配我们需要的额外的 x，但是如果全局匹配需要，回溯会逐个释放它们。所以它仍然会匹配过多内容：

```
/xx A xx/ foo() /xx B xx/
```

要解决这个问题，还得回到之前介绍的办法：准确表达我们希望表达的意思。如果我们说的“紧跟字符不是斜线的一些 x”其实就是除 x 之外的非斜线字符，就应该用「x+[^/x]」。它不会匹配‘…xxx/’，一连串 x 中表示注释结束的那个 x。事实上，它还有个问题，就是无法匹配注释结束之前的**任意多个** x，所以会在‘…xxx/’停下来。因为我们预计结束分隔符前只有一个 x，所以必须加入「x+/」处理这种情况。

于是得到「/x([^x]|x+[^/x])*x+/」，匹配最终的注释。

在自然语言和正则表达式之间翻译

第 273 页讨论用来匹配 C 注释“除结束分隔符之外的任何字符”的两种方法时，我提到两种办法：

x，之后的字符不是斜线：「x[^/]」
斜线，之前的字符不是 x：「[^x]/」

这种做法并不正式——自然语言的描述与正则表达式是非常不同的，你发现了吗？

要看这两者的差别，想象第一个表达式匹配字符串‘regex’的情况，最后的 x 之后没有斜线，但它不能被「x[^/]」匹配。字符组必须匹配一个字符，尽管这个字符不能是斜线，但它必须存在，可‘regex’中的 x 之后没有任何字符。第二个表达式的情况与此类似。当时，我们需要的正是符合这两个要求的表达式，所以自然语言的表述是错误的。

如果能使用顺序环视，“x，之后的字符不是斜线”可以直接写做「x(?!/)」。如果不能，就可以使用「x([^/]|$)」。它仍然需要匹配 x 之后的字符，但也可以匹配字符串的结尾。如果能够使用逆序环视，“斜线，之前的字符不是 x”就可以表示为「(?<!x)/」。如果不能，就需要使用「(^|[^x])/」。

在这个例子中，我们没有使用上述的任何一种办法，但知道有这些办法并不是坏事。

这看起来很迷惑，对吗？真正的表达式（用*取代 x）就是「/*([^*]|*+[^/*])**+/」，这样更复杂了，更不容易看懂，所以在理解复杂的正则表达式时，一定要保持清醒的思维。

消除 C 语言注释的循环

为了提高表达式的效率，我们必须消除这个表达式的循环。下一页的表 6-3 给出了我们能够“消除循环”的表达式。

和子域名的例子一样，「*normal**」必须匹配至少一个字符。子域名的例子中是因为 *normal* 部分不能为空。本例中必须的结束分隔符包含两个字符。我们确信，任何以结束分隔符的第一个字符结尾的任何 *normal* 序列，只有在紧跟字符不能组成结束分隔符的情况下，才会把控制权交给 *special* 部分。

表 6-3：消除 C 语言注释的循环

「opening normal* (special normal*)* closing」

元素	目的	正则表达式
opening	注释开始	/x
normal*	注释文本，包含一个或多个 'x'	[^x]*x+
special	不属于结束边界符的字符	[^/x]
closing	结尾的斜线	/

所以，按照通用的消除套路，我们得到：

```
/x[^x]*x+([^/x][^x]*x+)*/
```

请注意▲标注的位置。正则引擎可能有两种办法到达此处（267 页的表达式也是如此）。第一个是在开头的「/x[^x]*x+」匹配之后直接前进到此处，第二是在(…)*循环的某一轮中。无论哪种情况，只要到达此处，我们就知道已经匹配了 x，到达关键位置（pivotal point），可已经进入了注释的结尾分隔符。如果下面的字符是斜线，则匹配完成。如果是其他字符（当然不是 x），我们知道之前的判断是错误的，然后回到 *normal* 部分，等待下一个 x。找到之后我们再一次回到标记位置。

回到现实

「/x[^x]*x+([^/x][^x]*x+)*/」还不能直接拿来用。首先，注释是/*…*/而不是/x…x/。当然，我们可以很容易地把每个 x 替换为\x（字符组中的 x 替换为*）：

```
/\*[^*]*\*+([^/*][^*]*\*+)*/
```

实际情况中，注释通常会包括多行。如果匹配的注释包括多行，这个表达式也应该能够应付。如果是严格以行为处理单位的工具，例如 *egrep*，当然没办法用一个正则表达式匹配所有的行。对本书中提到的大多数工具，我们的确可以用这个表达式来匹配多行，删除它们。

在实际中，会遇到许多问题。这个正则表达式能够识别 C 的注释，但不能识别 C 语法的其他重要方面。例如，划线的部分尽管不是注释，也能够匹配：

```
const char *cstart = "/*", *cend = "*/"
```

我们会在下一节接着讨论这个例子。

流畅运转的表达式

The Freeflowing Regex

我们花了不少时间来构建匹配 C 的注释的正则表达式，但是没有考虑如何避免字符串中的错误匹配。使用 Perl 的话，你可能会想到用下面的程序过滤注释：

```
$prog =~ s{/\*[^*]*\*+(?:[^/*][^*]*\*+)*/}{}g;  # 去掉C语言注释(但有错误!)
```

表达式中，变量$prog 保存的文本会被删除（也就是，被空文本替换掉）。问题在于，如果在字符串内部找到注释的起始标记，正则表达式的匹配也不会停止，比如这段 C 代码：

```
char *CommentStart = "/*"; /* start of comment */
char *CommentEnd = "*/"; /* end of comment */
```

这里，下画线标注的部分是正则表达式的匹配结果，但是粗体标注的部分才是我们期望的。引擎寻找匹配时会在目标字符串的每个位置开始尝试匹配表达式。因为这种尝试只有在注释开始的地方（或者是看起来有可能开始的地方）成功，所以在大部分位置都无法匹配，传动装置的驱动过程继续向前，进入双引号字符串，其内容似乎是注释的开始。最好是能够告诉正则引擎，遇见双引号字符串时是应该尝试匹配还是直接跳过。当然，我们确实能做到这一点。

引导匹配的工具

A Helping Hand to Guide the Match

看下面的程序：

```
$COMMENT = qr{/\*[^*]*\*+(?:[^/*][^*]*\*+)*/};     #  匹配注释
$DOUBLE = qr{"(?:\\.|[^\\"])*"};                   #  匹配双引号字符串
$text =~ s/$DOUBLE|$COMMENT//g;
```

这里出现了两件新事物。其中之一是表达式「$DOUBLE|$COMMENT」，它由两个变量组成，都使用了 Perl 特有的 qr/…/正则表达式“双引号字符串”操作符。我们曾在第 3 章仔细讨论过（☞101），如果用字符串表示正则表达式，使用字符串的时候必须格外小心。Perl 提供的 qr/…/运算符解决了这个问题，它会把操作对象（operand）视为正则表达式，但不会实际应用它。我们在第 2 章（☞76）已经看到，这样非常方便。与 m/…/ 和 s/…/…/一样，我们可以自己选择分隔符（☞71），上面使用的是花括号。

另一点是通过用$DOUBLE 来匹配双引号字符串。传动装置驱动到$DOUBLE 能匹配的位置时，会一次性匹配整个双引号字符串。这里使用多选分支完全没有问题，因为二者之间并没有

重叠。如果我们从左向右扫描这个正则表达式就会发现，应用到字符串时，存在三种可能：

- 注释部分能够匹配，于是一次性匹配注释部分，直接到达注释的末尾，或者…
- 双引号字符串部分能够匹配，于是一次性匹配双引号字符串，直接到达其结尾，或者…
- 上面两者都不能匹配，本轮尝试失败。启动驱动过程，跳过一个字符。

这样，正则表达式永远不会从双引号字符串或者注释**内部开始**尝试，这就是成功的关键。实际上，到目前为止还不够，因为这个表达式在删除注释的同时也会删除双引号字符串，不过我们只需要再修改一小点就可以了。

```
$COMMENT = qr{/\*[^*]*\*+(?:[^/*][^*]*\*+)*/};     # 匹配注释
$DOUBLE = qr{"(?:\\.|[^\\"])*"};                   # 匹配双引号字符串
$text =~ s/($DOUBLE)|$COMMENT/$1/g;
```

唯一的区别在于：

- 设置了捕获型括号，如果能够匹配双引号字符串对应的多选分支，则$1 会保存对应的内容。如果匹配通过注释多选分支，$1 为空。
- 把 replacement 的值设置为$1。结果就是，如果双引号字符串匹配了，replacement 就等于双引号字符串——并没有发生删除操作，替换不会进行任何修改（不过存在伴随效应，即一次性匹配这个双引号字符串，直接到达其结尾，这就是把它放在多选结构首位的原因）。另一方面，如果匹配注释的多选分支能够匹配，$1 为空，所以会按照期望删除注释（注 8）。

最后我们还必须小心对付单引号的 C 常量，例如'\t'。这很容易——只需要在括号内添加另外一个多选分支。如果我们希望去掉 C++、Java、C#的//的注释，就可以把「//[^\n]*」作为第四个多选分支，列在括号外。

```
$COMMENT = qr{/\*[^*]*\*+(?:[^/*][^*]*\*+)*/};  # 匹配注释
$COMMENT2 = qr{//[^\n]*};                       # 匹配 C++ // 注释
$DOUBLE = qr{"(?:\\.|[^\\"])*"};                # 匹配双引号字符串
$SINGLE = qr{'(?:\\.|[^'\\])*'};                # 匹配单引号字符串

$text =~ s/($DOUBLE|$SINGLE)|$COMMENT|$COMMENT2/$1/g;
```

注 8：在 Perl 中，如果$1 在匹配中不能满足，则会获得一个特殊的“没有值”的值“undef”。用作 replacement 时，undef 就会被当作空字符串，程序运行的结果如我们所愿。不过如果打开了 Perl 的警报功能（每个优秀的程序员都应该这么做），这样使用 undef 会报警。为避免这种情况，应该在正则表达式之前使用编译指示“no warnings”，或者使用特殊的 Perl 替换符：

```
$text =~ s/($DOUBLE)|$COMMENT/defined($1)?$1 : ""/ge;
```

基本原理很好懂：引擎检查文本，迅速捕获（如果合适，则是删除）这些特殊结果。在我的老机器（配置大概停留在 1997 年的水平）上，Perl 脚本在 16.4 秒的时间内去掉了 16MB，500 000 行的测试文件中的注释。这已经很快了，不过我们仍然需要提高速度。

引导良好的正则表达式速度很快

A Well-Guided Regex is a Fast Regex

暂停一会儿，我们能够直接控制这个正则引擎的运转，进一步提高匹配速度。来考虑注释和字符串之间的普通 C 代码。在每个位置，正则引擎都必须尝试四个多选分支，才能确认是否能匹配，只有四个多选分支都匹配失败，它才会前进到下一个位置，这些复杂工作其实是不必要的。

我们知道，如果其中任何一个多选分支有机会匹配，开头的字符都必须是斜线、单引号或是双引号。这些字符并不能保证能够匹配，但是不满足这些条件绝对不能匹配。所以，与其让引擎缓慢而痛苦地认识到这一点，不如把「[^'"/]」作为多选分支，直接告诉引擎。实际上，同一行中任何数量的此类字符都能归为一个单元，所以我们使用「[^'"/]+」。如果你记得无休止匹配，可能会为添加的加号担心。确实，如果在某种(…)*循环中，它可能是很大的问题，但是在这个例子中，它完全没有问题（之后没有元素强迫它回溯），所以，添加：

```
$OTHER = qr{[^"'/]};        # 可能作为某个多选结构开头的字符
  ......
$text =~ s/($DOUBLE|$SINGLE|$OTHER+)|$COMMENT|$COMMENT2/$1/g;
```

出于某些我们即将要看到的原因，我把加号量词放在$OTHER 之后，而不是$OTHER 的内容之中。

我重新进行了性能测试，出乎意料的是，这样可以减少 75%的时间。通过改进，这个表达式节省了频繁尝试所有多选分支的大部分时间。仍然有些情况，所有多选分支都不能匹配（例如 'c▴/3.14'），此时，我们只能接受驱动过程。

不过，事情还没有结束，我们仍然可以让表达式更快：

- 在大多数情况下，最常用的多选分支可能是「$OTHER+」，所以我们把它排在第一位。POSIX NFA 没有这个问题，因为它总会检查所有的多选分支，但是对于传统型 NFA，它只要找到匹配就会停止，为什么不把最可能出现的多选分支放在第一位呢？

- 一个引用字符串匹配之后，在其他字符串和注释匹配之前，很可能出现的就是$OTHER的匹配。若在每个元素之后都添加「$OTHER*」，就能够告诉引擎下面必须匹配$OTHER，而不用马上进入下一轮/g 循环。

  这与消除循环的技巧是很相似的，此技巧之所以能提高速度，是因为它主导了正则引擎的匹配。这里我们使用了关于全局匹配的知识来进行局部优化，给引擎提供快速运转必须的条件。

  非常重要是，「$OTHER*」是加在每个匹配引用字符串的子表达式之后的，而之前的$OTHER（排在多选结构最前面的）必须用加号量词。如果你不清楚原因，请考虑下面的情况：添加的是$OTHER+，而某行中有两个连在一起的双引号字符串。同样，如果开头的$OTHER 使用星号量词，则任何情况都能匹配。

最终得到：

「($OTHER+|$DOUBLE$OTHER*|$SINGLE$OTHER*)|$COMMENT|$COMMENT2」

这个表达式能把时间再减少 5%。

回过头来想想最后两个改动。如果每个添加的$OTHER*匹配了过多的内容，开头的$OTHER+（我们将其作为第一个多选分支）只有两种情况下能够匹配：1）它匹配的文本在整个 s/…/…/g 的开头，此时还轮不到引用字符串的匹配；2）在任意一段注释之后。

你可能会想“从第二点考虑，我们不妨在注释后添加$OTHER+”。这很不错，只是我们希望用第一对括号内的表达式匹配所有希望保留的文本——不要把孩子连洗澡水一起倒掉。

那么，如果$OTHER+出现在注释之后，我们是否需要把它放在开头呢？我觉得，这取决于所应用的数据——如果注释比引用字符串更多，答案就是肯定的，把它放在第一位有意义。否则，我就会把它放后面。从测试数据来看，把它放在前面的效果更好。排在后面大约会损失最后的修改一半的效率。

完工

Wrapup

事情还没有结束。不要忘记，每个匹配引号字符串的子表达式都应该消除循环，本章已经花了很长的篇幅讲解这个问题。所以，最后我们要把这两个子表达式替换为：

```
$DOUBLE = qr{"[^\\"]*(?:\\.[^\\"]*)*"};
$SINGLE = qr{'[^'\\]*(?:\\.[^'\\]*)*'};
```

这样修改节省了 15%的时间。这些细小的修改把匹配的时间从 16.4 秒缩短到 2.3 秒，提升了 7 倍。

最后的修改还说明，用变量来构建正则表达式多么方便。`$DOUBLE` 可以作为单独的元素独立出来，可以改变，而不需要修改整个正则表达式。虽然还会存在一些整体性问题（包括捕获文本括号的计数），但这个技巧确实很方便。

这种便利是由 Perl 的 `qr/…/`操作符提供的，它表示与正则表达式相关的“字符串”。其他语言没有提供相同的功能，但是大多数语言提供了便于构建正则表达式的字符串。请参见 101 页的“作为正则表达式的字符串”。

下面是原始的正则表达式，看到它，你肯定会觉得上面的办法非常方便。为了便于印刷，我把它分为两行：

```
([^"'/]+|"[^\\"]*(?:\\.[^\\"]*)*"[^"'/]*|'[^'\\]*
(?:\\.[^'\\]*)*'[^"'/]*)|/\*[^*]*\*+(?:[^/*][^*]*\*+)*/|//[^\n]*
```

总结：开动你的大脑

In Summary: Think!

在本章的结尾讲个故事，我希望读者能够明白，在 NFA 中使用正则表达式时，稍微动动脑筋能带来多大的收益。在使用 GNU Emacs 时，我希望用一个正则表达式来找出某种类型的缩写，例如“don't”、“I'm”和“we'll”之类，同时必须忽略与单词邻接的单引号。我想用「`\<\w+`」来匹配单词，然后是「`'([tdm]|re|ll|ve)`」。这办法没有问题，但是我意识到，使用「`\<\w+`」是愚蠢的，因为这里只用到`\w`。你看到了，如果撇号之前就是一个`\w`，`\w+`显然也能够匹配，所以这个正则表达式检查并没有增加新的信息，除非我希望得到匹配的文本（在这里并不需要，我们只需要找到这个位置）。单独使用`\w`的正则表达式的速度是原来的 10 倍。

正因如此，一点点的思考就可以带来巨大的收获。我希望本章能够引发你的这点思考。



第  章

Perl

Perl

Perl 在本书中的分量很重，这样安排有充分的理由。Perl 很流行，提供的正则表达式特性很丰富，容易下载到，也很容易入门，而且在 Windows、Unix 和 Mac 等各种平台上都有提供。

Perl 的某些程序结构看上去类似 C 和其他传统编程语言，但也只是看上去像而已。Perl 解决问题的方式——**Perl 之道**（The Perl Way）——是不同于传统语言的。Perl 程序的设计通常使用传统的结构化和面向对象的概念，但是数据处理通常严重依赖正则表达式。我认为完全可以这么说：正则表达式在所有的 Perl 程序中都不可或缺。无论这个程序是 100 000 行，还是一行：

```
% perl -pi -e 's{([-+]?\d+(\.\d*)?)F\b}{sprintf "%.0fC",($1-32)*5/9}eg' *.txt
```

这个程序检查所有的 `.txt` 文件，将其中的华氏温度转换为摄氏温度（还记得第 2 章开头的例子吗）。

本章内容

本章讲解 Perl 的正则表达式的方方面面（注 1），包括正则流派的细节，和使用正则表达式的运算符。本章从基础开始介绍相关的细节，但我还是假设读者对 Perl 有基本的理解（如果你看过第 2 章，看本章就没多大问题）。那些没有详细讲解的细节，我会一笔带过，也不会费工夫来讲解语言中与正则表达式不相关的细节。在手边准备一本 Perl 的文档会很有帮助，或者 O'Reilly 的 *Programming Perl* 也行。

注 1：本书涵盖 Perl Version 5.8.5。

即使你目前对 Perl 还不够了解也不要紧，重要的是要有进一步学习的欲望。从任何方面来说，阅读本章都不是件轻松的事情。我的目的不是带读者入门，而是教给读者其他 Perl 的书中没提供的有用知识；为了保持本章内容的整体性和连贯性，我不会忽略一些重要的细节。某些问题很复杂，细节很多，如果不能马上理解也不必担心。我推荐读者第一遍阅读时只要了解全面的图景，需要的时候再返过来查阅。

下面列出了本章的结构作为指导：

- “Perl 的正则流派”（☞286）考察了 Perl 的正则表达式提供的丰富的元字符，以及正则文字提供的附加特性。
- “正则相关的 Perl 教义（Perlism）”（☞293）考察了在 Perl 中使用正则表达式的一些重要问题。详细介绍了“**动态作用域**（dynamic scoping）”和“**表达式应用场合**（expression context）”，并解释了它们与正则表达式之间的紧密联系。
- 正则表达式必须与应用方式结合起来才有价值，所以下面各节讲解了 Perl 中神奇的正则表达式控制结构：

  qr/…/运算符和 Regex 对象（☞303）
  Match 运算符（☞306）
  Substitution 运算符（☞318）
  Split 运算符（☞321）

- “巧用 Perl 的专有特性”（☞326）介绍了 Perl 独具的正则改良功能，包括在正则表达式的应用过程中执行任意 Perl 代码的功能。
- “Perl 的效率问题”（☞347）详细讲解了每个 Perl 程序员关注的问题。Perl 使用传统型 NFA 引擎，所以我们可以充分利用第 6 章介绍的各种技巧。当然，还有一些专属于 Perl 的问题会强烈地影响到 Perl 应用正则表达式的方式和速度。这些都会有所涉及。

前几章出现的 Perl

本书的大部分内容中都出现过 Perl：

- **第 2 章** 包括 Perl 的入门知识，给了许多例子。
- **第 3 章** 介绍了 Perl 的历史（☞88），用 Perl 语言介绍了许多应用正则表达式的问题，例如字符编码（包括 Unicode ☞105）、匹配模式（☞110），以及元字符（☞113）。

- **第 4 章** 解密了 Perl 使用的传统型 NFA 引擎。对 Perl 用户来说这一章非常重要。
- **第 5 章** 承接第 4 章，包含许多讨论过的例子。其中许多是以 Perl 给出的，即使有些例子不是以 Perl 给出的，它们的原理也适用于 Perl。
- **第 6 章** 对效率感兴趣的 Perl 程序员应该仔细阅读。

为了照顾不熟悉 Perl 的读者，前几章我都简化了 Perl 的例子，使用容易看懂的伪码。本章我会使用 Perl 风格的代码来举例。

作为语言组件的正则表达式

Regular Expressions as a Language Component

Perl 语言引人注目的特性之一就是，正则表达式在语言之中支持完美地内建。Perl 没有提供独立的正则表达式应用函数，它的正则表达式的运算符，包含在构成语言的其他丰富的运算符和结构之中。

Perl 具有强大的运用正则表达式的能力，人们可能认为，这需要数量繁多的运算符，但是，Perl 事实上只提供了四个与正则表达式有关的运算符，以及少量的相关元素（见表 7-1）。

表 7-1：Perl 中与正则表达式相关的对象概览

正则表达式相关运算符	修饰符	含义
m/*regex*/*mods* (☞306) s/*regex*/*replacement*/*mods* (☞318) qr/*regex*/*mods* (☞303) split(…) (☞321)	/x /o /s /m /i /g /c /e	正则表达式的解释方式（☞292、348） 引擎认定的目标字符串（☞292） 其他（☞311、315、319）
编译指示（Pragma）	**匹配完成之后的变量（☞299）**	
use charnames ':full'; (☞290) use overload; (☞341) use re 'eval'; (☞337) use re 'debug'; (☞361)	$1,$2 $^N $+ @- @+	捕获的文本 编号最小/最大的$1、$2… 表示目标字符串中
	$` $& $'	匹配之前、之中和之后的偏移值数组（最好不用，参见“Perl 的效率问题” ☞356）
相关函数	**相关变量**	
lc lcfirst uc ucfirst (☞290) pos (☞313) quotemeta (☞290) reset (☞308) study (☞359)	$_ $^R	默认的目标字符串（☞308） 内嵌代码的运行结果（☞302）

Perl 的功能非常强大，但它提供的运算符数量非常少，这样有利也有弊。

Perl 的长处

Perl's Greatest Strength

Perl 最大的优势可能在于，Perl 的运算符和函数提供了丰富的选项。根据应用场合的不同，它们的行为也不同，当然，这通常是执行者在那种场合自然想到的操作。O'Reilly 的 *Programming Perl* 说得很绝对："总的来说，Perl 的运算符可以做你希望的任何事情……"。正则匹配运算符 m/*regex*/提供了许多神奇的功能，会根据应用的场合、方式以及修饰符的不同而变化。

Perl 的短处

Perl's Greatest Weakness

表达能力太强，也是 Perl 最大的毛病之一。哪怕只是进行极小的修改，也有数不清的特殊情况、条件和场合在你眼皮底下发生变化，但却不会通知你——不经意之间就切换到另一种应用场合（注 2）。在 *Programming Perl* 这本书中，上面那句话的下半句是"只是缺乏一致性（consistency）"。当然，对计算机科学来说，固定、一致而值得依赖的接口是可取的。Perl 的强大功能在有经验的用户手里可能是强大的武器，但情况似乎是，你的 Perl 技能不断增长，是以不断地射伤自己的腿脚为代价的。

Perl 的正则流派

Perl's Regex Flavor

下一页的表 7-2 概要描述 Perl 的正则风格。以前，Perl 的许多元字符是其他系统不支持的，但是经过许多年之后，其他系统接受了许多 Perl 的创新。这些常见的特性在第 3 章的概览里有描述，但是 Perl 还有专属于自己的元素，会在本章后面讲解（表 7-2 覆盖了将要讲解的各个元素）

下面是对表格的补充：

① \b 只有在字符组内部才是退格符的简记法。在字符组外部，\b 表示单词分界符（☞133）。

八进制转义接收 2 到 3 位的数值。

「\x*num*」十六进制转义接收两位数字（也可以是一位数字，但是会报警）。「\x{*num*}」能接收任意长度的十六进制数。

注 2：虽然应用场合的数目很多，但我还是希望在本章涵盖所有内容。

表 7-2：Perl 的正则流派概览

字符组缩略表示法①	
115 (c)	\a [\b] \e \f \n \r \t *octal* \x*hex* \x{*hex*} \c*char*
字符组及相关结构	
☞118	字符组：[…] [^…]（可能包括类似 POSIX 的[:alpha:]表示法，☞127）
☞119	除换行符外的所有字符：点号（使用/s 时能匹配所有字符）
☞120	Unicode 组合字符序列：\X
☞120	单个字节（有危险）：\C
☞120 (c)	字符组缩略表示法②：\w \d \s \W \D \S
☞121 (c)	Unicode 属性，字母表和区块③：\p{*Prop*} \P{*Prop*}
锚点及其他零长度断言	
☞129	行/字符串起始位置：^ \A
☞129	行/字符串结束位置：$ \z \Z
☞315	前一次匹配的结束位置：\G
☞133	单词分界符④：\b \B
☞133	环视⑤：(?=…) (?!…) (?<=…) (?<!…)
注释和模式修饰符	
☞135	模式修饰符⑥：(?:*mods-mods*)，容许出现的字母是 x s m i（☞292）
☞135	模式修饰作用范围：(?*mods-mods*:…)
☞136	注释：(?#…) #…（若使用/x，则注释从‘#’开始，到行末结束）
分组、捕获、条件判断和控制	
☞137	捕获型括号：(…) \1 \2…
☞137	仅用于分组的括号：(?:…)
☞139	固化分组：(?>…)
☞139	多选结构：|
☞140	条件判断：(?*if then* | *else*)——if 部分可以为内嵌代码、环视，或是(*num*)
☞141	匹配优先量词：* + ? {n} {n,} {x,y}
☞141	忽略优先量词：*? +? ?? {n}? {n,}? {x,y}?
☞327	内嵌代码：(?{…})
☞327	动态表达式：(??{…})
专属于正则文字的功能	
☞289 (c)	变量插值：$*name* @*name*
☞290 (c)	大小写转换：\l \u
☞290 (c)	大小写转换范围：\U \L …\E
☞290 (c)	文字文本范围：\Q…\E
☞290 (c)	命名的 Unicode 字符：\N{*name*}——可选出现，参见第 290 页

(c) 表示可以在字符组内部使用 ①…⑥，参见补充内容

② \w、\d、\s 之类完全支持 Unicode。

Perl 的\s 不能匹配 ASCII 的垂直制表符（☞115）。

③ Perl 的 Unicode 支持针对的是 Unicode Version 4.1.0。

Unicode 字母表也能支持。字母表和属性名可以有‘Is’前缀，但并非必须（☞125）。区块名可以有‘In’前缀，但只有在区块和字母表的名字发生冲突时才必须使用。

Perl 也支持「\p{L&}」、「\p{Any}」、「\p{All}、「\p{Assigned}」和「\p{Unassigned}」属性。

Perl 支持例如「\p{Letter}」的长属性名。名字的各个单词之间可能是空格、下画线，或者什么也没有，（例如「\p{Lowercase_Letter}」，也可能写作「\p{Lowercase Letter}」）或者是「\p{Lowercase•Letter}」），为了前后一致，我推荐使用第 123 页表格中的长命名。

「\p{^…}」等价于「\P{…}」。

④ 单词分界符完全支持 Unicode。

⑤ 顺序环视可能包含捕获型括号。

逆序环视中的子表达式必须匹配固定长度的文本。

⑥ /x 修饰符只能识别 ASCII 空白字符。/m 只对换行符有影响，而且不是所有的 Unicode 换行符。

/i 能够在 Unicode 中正常工作。

所有的元字符并不是生而平等的。“正则元字符”没有得到正则引擎的支持，但 Perl 的正则文字预处理机制能对付。

正则运算符和正则文字

Regex Operands and Regex Literals

表 7-2 最下面的条目标注有“专属于正则文字”。**正则文字**（regex literal）就是 m/***regex***/部分中的“***regex***”，虽然平时称其为“正则表达式”，但在‘/’分隔符之间的部分是有自己的解析规则。用 Perl 的行话来说，正则文字就是“表示正则含义的双引号字符串（regex-aware double-quoted string）”，及处理之后传递给正则引擎的结果。正则文字处理机制提供了特殊的功能来构建正则表达式。

举例来说，正则文字提供了**变量插值功能**。如果变量$num 的值是 20，代码 m/:.{$num}:/得到的就是「:.{20}:」。这样可以根据需要即时构建正则表达式。正则文字的另一功能是大

小写自动切换展开，`\U…\E` 可以保证其中的字母均为大写。比如，`m/abc\Uxyz\E/`得到正则表达式「abcXYZ」。这个例子有点做作，如果需要使用「abcXYZ」，应该直接输入 `m/abcXYZ/`，但是这种功能结合变量插值就很有用：如果变量`$tag` 包含字符串“title”，则代码 `m{</\U$tag\E>}` 得到「</TITLE>」。

除正则文字之外还有什么呢？我们可以把字符串（或者任何表达式）当作正则运算元，比如：

```
$MatchField = "^Subject:";  #  普通字符串赋值
  ......
if ($text =~ $MatchField) {
    ......
```

当`$MatchField`用作`=~`的运算元时，它的值就被解释（interpreted）为正则表达式。这里只能“解释”普通的正则表达式，所以不支持只作用于正则文字的变量插值和「`\Q…\E`」。

下面的例子值得思考，如果把：

```
$text =~ $MatchField
```

替换为：

```
$text =~ m/$MatchField/
```

结果完全一样。这里的正则文字只包含一个元素——变量`$MatchField`。正则文字中插值变量的值不会被当作正则文字处理，所以变量内的`\U…\E` 和`$var` 之类不会被识别（第 292 页说明了正则文字的处理细节）。

如果正则表达式在程序的执行期间需要多次用到，那么正则运算元采用字符串或变量插值的效率差距就很明显。第 348 页讨论了这个问题。

正则文字支持的特性

正则文字提供下面的特性：

- **变量插值** 正则表达式中以`$`和`@`开头的变量会被替换为实际变量的值。`$`变量插入一个简单的纯量值（scalar value）。以`@`开头的插入数组或者数组的一部分，以空格分隔各个元素（其实是以`$"`变量作分隔符，它的默认值是空格）。

  在 Perl 中，‘`%`’引入一个散列变量（hash variable），但是在字符串中插入一个散列变量并没有太大的意义，所以 Perl 不支持`%`插值。

- **命名的 Unicode 字符** 如果程序中包含“**use charnames ':full';**”，就可以用\N{*name*}序列引用 Unicode 字符。例如，\N{LATIN SMALL LETTER SHARP S}匹配“ß”。在 Perl 的 *unicore* 目录下的 *UnicodeData.txt* 中可以找到 Perl 支持的 Unicode 字符列表。下面的代码能够报告文件的位置：

```
use Config;
print "$Config{privlib}/unicore/UnicodeData.txt\n";
```

  “**use charnames ':full';**”很容易忘记，或者忘记在‘full’前面添加冒号，果真如此的话，\N{…}就不能正常工作。同样，如果使用了下面介绍的正则表达式重载，\N{…}也不能正常工作。

- **大小写转换前缀 \l** 和**\u** 能够把后面的字符转换为小写或大写形式。通常我们使用此功能来转换插值变量的第一个字符。举例来说，如果变量$title 包含“mr.”，m/…\u$title…/就能生成正则表达式「…Mr.…」。Perl 的 lcfirst()和 ucfirst()函数提供了同样的功能。

- **大小写转换范围 \L** 和**\U** 能够把后面所有的字符转换为小写或大写，其作用范围一直到表达式末尾，或是**\E** 为止。同样是$title，m/…\U$title\E…/会产生正则表达式「…MR.…」。Perl 的 lc()和 uc()函数提供了同样的功能。

  我们可以把这两者结合起来：无论变量$title 采用怎样的字母组合，m/…\L\u$title\E…/都会得到「…Mr.…」。

- **文字文本范围 \Q**“转义（quote）”正则表达式元字符（也就是在它们之前放一个反斜线，保证它们只作为普通的字符），其作用范围直到字符串的结尾，或者直到**\E**。它能转义正则表达式元字符，但不能转义表示变量插值的正则文字**\U**，当然也不能转义**\E**。奇怪的是，如果反斜线开头的字符序列不能识别——例如\F 或者\H，反斜线也不会被转义。即使是\Q…\E，这样的序列也会导致“unrecognized escape”警告。

  在实践中，这些限制并不是严重的缺陷，\Q…\E 通常用于引用插值文本，这样就可以正确转义所有的元字符。例如，如果$title 包含“Mr.”，那么代码 m/…\Q$title\E…/就会生成正则表达式「…Mr\.…」，我们要的就是这样——希望匹配的是$title 中的**字符**，而不是$title 中的**正则表达式**。

  如果你希望在正则表达式中包含某些用户输入的数据，这非常有用。举例来说，m/\Q$UserInput\E/i 能够对$UserInput（作为字符串，而不是正则表达式）中的字符进行不区分大小写的搜索。

  Perl 的函数 quotemeta()提供了与\Q…\E 等价的功能。

- **重载** 借助重载，用户可以使用任何期望的方式预处理正则文字的文字字符。这是概念值得讨论，但是目前的实现还有诸多限制。关于重载的细节讨论请参见第 341 页。

使用自己的正则表达式分隔符

Perl 语法中最奇妙（也是最有用）的特性之一就是用户可以使用自己的正则文字分隔符。传统的分隔符是斜线，例如 m/…/、s/…/…/和 qr/…/，不过还可以使用除数字、字母和空格之外的字符。常用的包括：

```
m!…!            m{…}
m,…,            m<…>
s|…|…|          m[…]
qr#…#           m(…)
```

右边四个是特殊的分隔符：

- 右边的四个例子具有不同的开始-结束分隔符，而且可能嵌套（也就是说，如果开始-结束分隔符匹配恰当，表达式中容许包含与分隔符一样的字符）。因为圆括号和方括号在正则表达式中经常用到，m(…)和 m[…]可能不如其他更有吸引力。使用/x 修饰符时，可能出现下面的形式：

  ```
  m{
     regex  # comments
     here   # here
  }x;
  ```

  也可以使用某种组合标记 regex，另一组（如果你喜欢，也可以用同样的）标记 replacement。例如：

  ```
  s{…}{…}
  s{…}!…!
  s<…>(…)
  s[…]/…/
  ```

  如果这样做了，就可以在两对分隔符之间插入空格和注释。第 319 页进一步讲解了 substitution 运算符的 replacement 运算元。

- 对 match 运算符来说，把问号作为分隔符有其特殊价值（禁止更多的匹配），这一点在下一节讲解关于 match 运算符时讨论（☞308）。

- 288 页已经提到，正则文字被解析成“表示正则含义的双引号字符串”。如果用单引号作分隔符，就无法使用这些功能。使用 m'…'时就不会进行变量插值，实时修改文本的结构（比如\Q…\E）不会生效，\N{…}也无法使用。也许在使用包含多个@的正则表达式时 m'…'很有价值，因为这样可以不需要转义。

如果进行 match 操作，而分隔符是斜线或者问号，可以省略 m，也就是：

```
$text =~ m/…/;
$text =~  /…/;
```

是等价的。但我更喜欢明确写上 m。

正则文字的解析方式

How Regex Literals Are Parsed

大多数情况下，如果用户“只会用到”上文讲解的正则文字特性，就不需要理解 Perl 将它们转换为真正的正则表达式的具体细节。就这一点来说，Perl 直观性非常方便，但是许多时候，了解细节并无坏处。下面列出了各种处理的顺序：

1. 找到结束分隔符，读入修饰符（例如/i 之类）。下面的处理就能判断是否采用了/x 之类的模式。

2. 变量插值。

3. 如果使用了正则表达式重载，正则文字的每个部分都会交给重载子程序来处理。各部分由插值变量分隔；插入的值是无法重载的。

   如果正则表达式没有进行重载，处理\N{…}。

4. 应用大小写转换结构（例如\Q…\E）。

5. 把结果提交给正则引擎。

以上是程序员眼中的处理，但是 Perl 内部的处理其实是很复杂的。单单是第二步，就必须识别正则表达式的元字符，比如不应把「this$|that$」下画线的那部分识别为变量。

正则修饰符

Regex Modifiers

Perl 的正则运算符容许在正则文字的结束分隔符之后添加正则修饰符（例如 m/…/**i**、s/…/…/**i** 和 qr/…/**i** 中的 **i**）。所有运算符都支持的核心修饰符一共有 5 种，详见表 7-3。

头四种在第 3 章已经介绍过，它们能够作为模式修饰符（☞135）或者范围模式修饰符（☞135），在正则表达式之中使用。如果正则表达式内部出现了修饰符，match 运算符也用

表 7-3：所有正则运算符可用到的核心修饰符

/i	☞110	进行忽略大小写的匹配
/x	☞111	宽松排列和注释模式
/s	☞111	点号通配模式
/m	☞112	增强的行锚点模式
/o	☞348	仅编译一次

到了修饰符，则正则表达式内部的修饰符的优先级更高（从另一方面来说就是，一旦修饰符应用到正则表达式内部的某些元素，这些元素就不再受其他修饰符的影响）。

第五个核心修饰符/o，与效率有很大的关系。此问题从第 348 页开始讨论。

如果需要使用多个修饰符，只需要把它们并排列在结束分隔符之后即可，排列的顺序是无关紧要的（注 3）。请注意，斜线本身不是修饰符，你可以使用 **m/**<title>**/i**、**m|**<title>**|i**，或是 **m{**<title>**}i**，甚至是 **m<**<title>**>i**。不过在讲解修饰符时，通行的做法是加上一个斜线，例如“修饰符/i”。

正则表达式相关的 Perl 教义

Regex-Related Perlisms

学习正则表达式，还需要掌握许多一般的 Perl 概念。下面几节的内容包括：

- **应用场合**（context）Perl 的重要概念之一就是，许多函数和运算符在不同应用场合有不同的意义。例如，Perl 的 while 循环希望接收一个纯量值（scalar value）作为判断条件，但对于 print 语句希望接收一组值（a list of value)。因为 Perl 容许表达式对其应用场合进行“响应”(respond)，同样的表达式在不同的应用场合可能得到截然不同的结果。
- **动态作用域**（dynamic scope）大多数编程语言都支持本地变量和全局变量，但是 Perl 还提供了另一种复杂功能，称为动态作用域。动态作用域会临时“保护”全局变量，保存一份副本，稍后自动恢复。这个复杂的概念对我们来说很重要，因为它影响到$1 和其他的匹配相关变量。

注 3：因为修饰符可以以任意形式出现、程序员通常会花很多精力调整修饰符，让它们看上去最顺眼。learn/by/osmosis 是没问题的(假设函数名是 learn)，其中的修饰符是 osmosis。修饰符可以重复出现，但这没有意义（稍后讨论的替换运算符的/e 是个例外)。

表达式应用场合

Expression Context

context 对 Perl 来说是很重要的概念，尤其对 match 运算符来说更是如此。一个表达式可能出现在三种 context 中：**序列**（list）、**纯量值**（scalar）或者**空**（void），它们表示表达式期望接收的参数类型。所以，*list context* 说明表达式期望获得一个序列。*scalar context* 说明表达式期望获得单个值。以上两者极为常见，而且对使用正则表达式非常有价值。*void context* 说明不期望获得任何值。

看下面两个赋值：

```
$s = expression one;
@a = expression two;
```

因为`$s`是 scalar 变量（它用来保存单个的值，而不是序列），期望简单的纯量值，所以第一个表达式的应用场合为 scalar context。同样，因为`@a`是一个数组，期望获得一个 list，第二个表达式的应用场合为 list context。即使这两个表达式完全等价，也可能返回完全不同的结果，产生不同的影响。具体情况依表达式而定。

举例来说，`localtime` 函数如果用在 list context 中，会返回一组值，表示当前年、月、日、时。但如果用在 scalar context 中，则返回文本类型的当前时间，比如‘`Mon Jan 20 22:05:15 2003`’。

另一个例子是`<MYDATA>`之类的 I/O 运算符，在 scalar context 中，它返回文件的下一行，但是在 list context 中，返回所有（剩下的）行。

就像 `localtime` 和 I/O 运算符一样，许多 Perl 的结构会根据应用场合的不同返回不同的值，正则运算符同样如此。拿 match 运算符 `m/…/`来说，有时候它会简单地返回 true/false 值，有时候返回一组匹配结果。所有的细节都会在本章讲解。

强转正则表达式

不是所有的正则表达式天生都能区分场合的，所以，如果某个应用场合中正则表达式无法提供期望的返回类型，就要按照 Perl 的规定处理。为了把方桩插入圆孔，Perl 会“强转（contort）”这个值。如果在 list context 中返回的是 scalar 值，Perl 会生成只包含单个元素的 list。这样`@a = 42`就等于`@a = (42)`。

另一方面，把 list 转换为 scalar 却没有统一的规定。如果程序是这样：

```
$var = ($this, &is, 0xA, 'list');
```

逗号运算符返回最后的元素‘list’给$var。如果给定的是一个数组，例如**$var = @array**，则返回数组的长度。

其他语言用不同的术语描述这种处理，例如**修正**（cast）、**提示**（promote）、**强制转换**（coerce）或**转换**（convert），但是我认为这些词都已经具有了自己的意义（有点令人讨厌），不适合描述 Perl 的做法，所以我使用“强转（contort）”。

动态作用域及正则匹配效应

Dynamic Scope and Regex Match Effects

Perl 的变量分为两类（全局变量和私有变量），动态作用域是正确理解它们的重点，研究正则表达式时也需要关注此概念，因为它关系到匹配完成之后信息如何使用。下一节介绍了这些概念及其与正则表达式的关系。

全局和私有变量

总的来说，Perl 提供了两种变量：全局的和私有的。私有变量使用 my(…)来声明，全局变量不需要声明，在使用时会自动出现。全局变量通常在程序的任何地方都是可见的，而私有变量，按照语言的规定只有在它们所属的代码块之内才是可见的。也就是说，只有私有变量声明所在的代码块之内的 Perl 代码，能够访问私有变量。

全局变量的使用则很普通，只是有的特殊变量不太好理解，例如$1、$_、@ARGB 之类。普通用户的变量是全局的，除非它们以 my 来声明，否则即使它们“看上去”是私有的，也是全局变量。按照 Perl 的规定，Package Acme::Widget 中的全局变量$Debug，虽然有**完整的限定名**$Acme::Widget::Debug，仍然是一个全局变量。如果出现了 **use strict;**，则所有（不包括特殊的）全局变量必须使用完整的限定名，或者通过 our 来声明（our 声明一个**名称**（name），而不是一个新变量，请参考 Perl 的文档）。

使用动态作用域的值

动态作用域（dynamic scoping）是个值得一提的概念，很少有编程语言提供这种功能。下文会讲解它与正则表达式的关系，简单地说，动态作用域可以让 Perl 保存全局变量的一个副

本，在某个代码块中修改此副本，退出之后自动恢复原来的值。保存副本的操作就称为**生成动态作用域**（creating a new dynamic scope），或者**本地化**（localizing）。

使用动态作用域的原因之一是为了临时改变某些保存在全局变量中的某些全局状态。举例来说，package Acme::Widget 提供了一个调试标志位（flag），我们可以修改全局变量 $Acme ::Widget::Debug 来启用或者停用调试功能。下面的代码可以临时改变此标志位：

```
  ......
{
  local($Acme::Widget::Debug)  = 1; #  确保启用
  #  此时 Acme::Widget::Debug 已启用、可以调试
     ......
}
  #  $Acme::Widget::Debug 现在恢复到原来的值
```

local 函数的命名很成问题，但它生成了一个新的动态作用域。**调用 local 并没有创造新的变量**，local 是行为，而不是声明。在全局变量之前，local 做了三步处理：

1. 在内部保存变量值的副本；

2. 把新值赋予到变量（无论是 undef 还是传给 local 的值）；

3. local 代码块执行结束之后，把变量恢复到之前的值。

也就是说，"local" 指的是对变量的修改的持续时间。对本地化的变量来说，持续时间就是代码块执行的时间。如果代码块中调用了子程序，本地化的值仍然保留（毕竟，变量仍然是一个全局变量）。它与非本地化的全局变量的唯一区别是，在代码块执行完成之后，之前的值会被恢复。

local 对全局变量的自动保存和恢复比想象的要复杂。请参考表 7-4 右侧，详细了解背后的处理。

为方便起见，我们也可以给本地变量赋一个值 local($SomeVar)，这等于把 undef 赋值给 $SomeVar。如果不使用括号，表示强制使用 scalar context。

举个实际的例子，我们需要调用一个写得很糟糕的函数，而它会产生许多"Use of uninitialized value"的警告。优秀的 Perl 程序员都会使用 Perl 的-w 选项来解决这个问题，但是库的作者

表 7-4：local 的含义

<table>
<tr><th>普通 Normal 程序</th><th>等价程序</th></tr>
<tr><td><pre>{
    local($SomeVar); # save copy

    $SomeVar = 'My Value';
        ......

} # 自动恢复到之前的值</pre></td><td><pre>{
    my $TempCopy = $SomeVar;
    $SomeVar = undef;

    $SomeVar = 'My Value';
        ......
    $SomeVar = $TempCopy;
}</pre></td></tr>
</table>

显然没有。你对这些警告非常恼火，但是如果不能修改程序库，有什么其他简便办法来代替-w 吗？这时候可以使用对$^W 的调用的 local，即时关闭警报（^W 可以表示为两个字符，脱字符和 'W'，也就是 ctrl+W)。

```
{
    local $^W = 0;      # 确保关闭警报
    UnrulyFunction(…);
}
# 退出代码块，把$^W 恢复到原来的值
```

无论全局变量$^W 是什么值，调用 local 保存都会为其保存一份内部副本。然后$^W 被用户置为 0。上面提到的糟糕程序在执行时，Perl 检查$^W，发现其值为 0，就不会发出警报。在函数返回时，新值 0 仍然有效。

这样看来，不用 local 的话似乎也没有问题。不过，在子程序返回，代码块退出时，$^W 会恢复到之前的值。这种改变是本地的、**即时的**，只在代码块内部生效。按照表 7-4 右侧的做法，用户可以手工生成和返回副本，达到同样的效果，但是 local 更为方便。

考虑在其他情况下会发生什么，比如用 my 替代 local（注 4）。my 会新建一个变量，其初始值是 undef。只有在声明的代码块中才可见（也就是说，在 my 和它所在的代码块结束之间）。它不会改变、修改，或以其他方式引用和影响其他变量，包括可能存在的同样名字的全局变量。新建的变量在程序的其他部分都不可见，包括在那个糟糕的程序内。这样新的$^W 的确被置为 0，但永远不会再使用或者引用，所以它完全是白费工夫（执行糟糕的程序时，Perl 根据与其无关的全局变量$^W 决定是否报警）。

注 4：Perl 不容许对这个特殊的变量名使用 my，所以这种比较只有理论意义。

更好的比喻：充分的透明度

可以这样理解 local，它对变量的修改是用户完全无法察觉的（好像是把新值投影到原变量之上）。用户（还包括能看到的任何人，例如子程序和信号处理程序）会看到这些新的值。在代码块结束之前，local 的修改会取代之前的值。退出之后，这种透明特性会自动消除，也就是取消 local 进行的所有修改。

相比“保存一个内部副本”，这个比喻更接近现实。使用 local 并不会生成一个副本，而是在访问变量时，使用新设置的值（即屏蔽原来的值）。退出代码块之后会抛弃新设置的值。调用 local 时，新值是手动设置的，但我们要讲解这些细节的原因在于：正则表达式的伴随效应变量（side-effect variables）会自动使用动态作用域。

正则表达式的伴随效应和动态作用域

正则表达式与动态作用域有什么关系呢？关系很大。作为伴随效应，许多变量——例如$&（引用匹配的文本）和$1（引用第一组括号内表达式匹配的文本）——会在匹配成功时自动设置。在下一节会详细讨论这些问题。在其所处的代码块中，这些变量都会自动使用动态作用域。

这种设计的好处在于，每次调用子程序都要启动新的代码块，也就是为这些变量提供了新的动态作用域范围。因为在代码块之前的值会在代码块执行完之后恢复（也就是子程序返回时），子程序不能改变调用方能看到的值。

来看个例子：

```
if ( m/(…)/ )
{
    DoSomeOtherStuff();
    print "the matched text was $1.\n";
}
```

因为$1 的值在进入代码块时进行了动态作用域处理，这段代码不关心也不必关心，函数 DoSomeOtherStuff 是否改变了$1 的值。此函数对$1 的任何改动都只在函数定义的代码块内部，或者函数的子代码块中生效。所以，DoSomeOtherStuff 不会影响 print 接收的$1 的值。

自动使用动态作用域很有用，虽然有时候不那么明显：

```
if ($result =~ m/ERROR=( .*)/) {
   warn "Hey, tell $Config{perladmin} about $1!\n";
}
```

标准库模块 Config 定义了一个关联数组（associative array）%Config，其成员$Config-{perladmin}保存本地 Perlmaster 的 E-mail 地址。如果$1 没有使用动态作用域，这段代码就很难理解，因为%Confg 是一个**绑定**变量（tied variable）。也就是说，对它的任何引用都意味着幕后的子程序调用，用$Config{…}进行正则表达式匹配时，Config 中的子程序返回对应的值。这次匹配发生在上一行的匹配和对$1 的使用中间，所以如果$1 没有使用动态作用域，它的值会被修改。所以，$Config{…}中对$1 的任何修改都被动态作用域安全地保护了起来。

动态作用域还是词法作用域

如果使用恰当，动态作用域能提供许多便利，但是滥用动态作用域会带来无休止的噩梦，因为阅读程序的人很难理解，分散在散落的 local、子程序和本地变量引用之间的复杂交互。

我曾说，my(…)声明会在**词法范围**（lexical scope）内创造一个私有变量。与私有变量的词法范围对应的是全局变量的范围，但是词法范围与动态作用域没有关系（仅有的联系是：不能对 my 变量调用 local）。请记住，local 只是**行为**（action），而 my 既是行为，又是声明，这很重要。

匹配修改的特殊变量

Special Variables Modified by a Match

成功的匹配会设置一组只读的全局变量，它们通常会自动使用动态作用域。如果匹配不成功，这些值**永远**也不会改变。在需要的时候，它们会设置为空字符串（不包括任何字符的字符串）或者 undefind（“未定义”，一个“没有值”的值，与空字符串类似，但测试时两者不相等）。表 7-5 给出了若干例子。

详细地说，匹配完成之后会设置这些变量：

**$&** 正则表达式所匹配文本的副本。从效率方面考虑（参见第 356 页的讨论），最好不要使用这个变量（还包括下面介绍的$` 和$' ）。一旦匹配成功，$&就不会是未定义状态，尽管它可能是空字符串。

表 7-5：匹配后特殊变量的说明

如下匹配完成之后

```
                          12                 2 3   4    4 31
"Pi is 3.14159, roughly" =~ m/\b((tasty|fattening)|(\d+(\.\d*)?))\b/;
```

设置了下面的特殊变量

变量	含义	值
$`	匹配文本之前的文本	Pi•is•
$&	匹配文本	3.14159
$'	匹配文本之后的文本	,•roughly
$1	第 1 组括号匹配的文本	3.14159
$2	第 2 组括号匹配的文本	*undef*
$3	第 3 组括号匹配的文本	3.14159
$4	第 4 组括号匹配的文本	.14159
$+	编号最大的括号匹配的文本	.14159
$^N	最后结束的括号匹配的文本	3.14159
@-	目标文本中各匹配开始位置的偏移值数组	(6, 6, undef, 6, 7)
@+	目标文本中各匹配结束位置的偏移值数组	(13, 13, undef, 13, 13)

**$`** 在目标文本中匹配开始之前（左边）文本的副本。如果使用/g 修饰符，你可以期望$ 的起点是**开始尝试位置**的文本，但它每次都是从整个字符串的开始位置开始的。如果匹配成功，$`肯定不会是未定义状态。

**$'** 保存目标文本中匹配成功文本之后（右边）的文本的副本。如果匹配成功，$'肯定不会是未定义状态。匹配成功之后，字符串"$` $& $'"就是目标字符串的副本（注 5）。

$1、$2、$3、…

对应第 1、2、3 组捕获型括号匹配的文本（请注意，这里没有$0，因为它是脚本的名字，与正则表达式无关）。如果它们对应的括号在表达式中不存在，或者没有实际参与匹配，则设置为未定义状态。

匹配之后就可以使用这些变量，在 s/…/…/中的 replacement 也可以使用。它们还能在动态正则结构或者嵌入代码中使用（☞327）。在正则表达式中使用这些变量是没多少意义的（因为已经有了「\1」之类）。请参考第 303 页的“在正则表达式中使用$1”。

「(\w+)」和「(\w)+」的区别可以用来说明$1 的设置方式。两个表达式都能匹配同样的文

注 5：事实上，即使目标字符串是未定义的，但能匹配成功（虽然不太现实，但有可能），"$` $& $'"是一个空字符串，而不是未定义。只有在这种情况下，两者才不一样。

本，但是它们的区别在于括号内的子表达式匹配的内容。用这个表达式匹配字符串‘tubby’，第一个表达式的$1 的内容是‘tubby’，而第二个表达式中的$1 只包含‘y’：在「(\w)+」中，加号在括号外面，所以每次迭代都会重新捕获，$1 保留最后的字符。

还需要注意的是「(x)?」和「(x?)」的差别。前一个表达式中括号及其捕获内容不是必然出现的，所以$1 可能是‘x’，或者是未定义，但「(x?)」中括号在匹配的外面——匹配的内容不是必然出现的，但匹配必须发生。如果整个表达式匹配成功，这部分的匹配必然会发生，尽管「x?」匹配的是空字符串。所以在「(x?)」中，$1 可能是‘x’或者是空字符串。下表给出了一些例子：

实例	$1	实例	$1
"::" =~ m/:(A?):/	空字符串	"::" =~ m/:(\w*):/	空字符串
"::" =~ m/:(A)?:/	未定义	"::" =~ m/:(\w)*:/	未定义
":A:" =~ m/:(A?):/	A	":Word:" =~ m/:(\w*):/	Word
":A:" =~ m/:(A)?:/	A	":Word:" =~ m/:(\w)*:/	d

从上表可以看出，如果需要添加括号来捕获文本，如何添加取决于我们的意图。在所举的例子中，增加的括号对整体匹配没有影响（整体匹配是不变的），其中唯一的区别就是$1 设置的伴随效应。

$+ 表示$1、$2 等匹配过程中明确设定的，编号最大的变量的副本。在下面的情况中会有用：

```
$url =~ m{
   href \s* = \s*        #  匹配"href = "，然后是它的值...
   (?: "([^"]*)"         #  双引号字符串，或者...
     | '([^']*)'         #  单引号字符串，或者...
     |  ([^'"<>]+) )     #  非引号形式的值
}ix;
```

如果没有$+，我们可能需要依次检查$1、$2 和$3，才能找出明确设置的那个。

如果正则表达式中没有捕获型括号（或者在匹配中没有用到），则这个值为未定义。

**$^N** 最后结束的，在匹配中明确设定的括号匹配的文本的副本（明确设定的$1 等变量中，闭括号在最后）。如果正则表达式中没有捕获型括号（或者匹配中没有用到），则其值为未定义。第 344 页开头有个恰当的例子。

@-和@+

表示各捕获型括号所匹配文本的起始和结束位置在目标文本中偏移值的数组。使用起来可能有点迷惑，因为它们的名字比较怪异。两个数组的第一个元素都对应整体匹配。也就是说，通过$-[0]访问到的@-的第一个元素，是整个匹配在目标字符串中的偏移值，即：

```
$text = "Version 6 coming soon?";
......
$text =~ m/\d+/;
```

$-[0]的值为8，代表匹配从目标字符串的第8个位置开始（在Perl中，偏移值从0开始）。

@+的第一个元素通过$+[0]访问，表示对应匹配文本结束位置的偏移值。在上例中其值为9，表示整体匹配结束于目标字符串的第9个字符之前。所以，如果$text没有变化，**substr($text, $-[0], $+[0] - $-[0])**就等于$&，但没有$&那样的性能缺陷（☞356），下例给出了@-的简单用法：

```
1 while $line =~ s/\t/' ' x (8 - $-[0] % 8)/e;
```

它会把给定文本中的制表符（tab）替换为合适长度的空格序列（注6）。

两个数组中接下来的元素分别对应各捕获分组的开始位置和结束位置的偏移值。$-[1]和$+[1]对应$1，$-[2]和$+[2]对应$2，依次类推。

**$^R** 这个变量保存最近执行的嵌入代码的结果，如果嵌入代码结构作为「(? *if then* | *else*)」条件语句（☞140）中的if部分，则不设定$^R。在正则表达式内部（即在嵌入代码或者动态正则结构☞327中），它会自动根据匹配的各部分进行本地化处理，所以因为回溯而“交还”的代码对应的$^R的值会被放弃。换一种说法就是，它保存引擎到达当前状态的工作路径中“最近”的值。

如果正则表达式根据/g修饰符重复使用，那么每次循环都会重新设置这些变量。也就是说可以在s/…/…/g中使用$1，因为每次匹配时它的值都不一样。

注6：这段代码的局限在与，它只能处理“传统”西方文本，而无法正确处理包含“枝”之类的宽字符集，因为宽字符集中显示一个字符可能需要两个以上的位置，某些Unicode语音字符，例如à，也无法处理（☞107）。

在正则表达式中使用$1

Perl 手册专门提到，在正则表达式外部，不能用「\1」反向引用（而应该使用$1）。变量$1 对应上次成功匹配中的某个固定字符串。「\1」则是正则表达式元字符，它对应**正则引擎遇到**「\1」时第一组捕获型括号捕获的文本。在 NFA 的回溯过程中，它的值可能会变化。

与之对应的问题是，正则运算元中是否能够使用$1 之类的变量。通常，在内嵌代码或者动态正则结构（☞327）中可用，在其他情况下就没什么意义。出现在运算元中“表达式部分”的$1 与其他变量一样处理：在匹配或替换操作开始时插值。也就是说，对正则表达式而言，$1 与当前的匹配没什么关系，它属于上一次匹配。

qr/…/运算符与 regex 对象

The qr/…/ Operator and Regex Objects

在第 2 章和第 6 章已经简要介绍过（☞76，277）一元运算符 qr/…/，其运算元为正则表达式，返回 regex 对象。返回的对象可以被之后的正则运算符用来匹配、替换、分割，或者可以作为其他的更长表达式的一部分。

regex 对象主要用来把正则表达式封装为一个单元，构建大的正则表达式，以及提高效率（在正则表达式编译时提高控制能力，讨论见下文）。

291 页已经介绍过，用户可以使用自己的分隔符，例如 qr{…}或者 qr!…!。它还支持核心修饰符/i、/x、/s、/m 和/o。

构建和使用 regex 对象

Building and Using Regex Objects

下面的表达式来自第 2 章（☞76）：

```
my $HostnameRegex = qr/[-a-z0-9]+(?:\.[-a-z0-9]+)*\.(?:com|edu|info)/i;

my $HttpUrl = qr{
   http:// $HostnameRegex \b                       #  Hostname
   (?:
       / [-a-z0-9-:\@&?=+,.!/~*'%\$]*              #  可能出现的path
          (?<![.,?!])                              #  不容许以[.,?!]结尾
   )?
}ix;
```

第一行代码把匹配主机名所用的简单正则表达式封装为regex对象，保存到变量$HostnameRegex中。下一行使用该变量构建匹配HTTP URL的regex对象，保存到$HttpUrl中。构建完成之后就可以以多种方式使用，例如：

```
if ($text =~ $HttpUrl) {
  print "There is a URL\n";
}
```

用来检测，或者：

```
while ($text =~ m/($HttpUrl)/g) {
  print "Found URL: $1\n";
}
```

用来搜索和显示所有的HTPP URL。

如果按照第5章的讲解（☞205）修改$HostnameRegex：

```
my $HostnameRegex = qr{
   #  一个或多个点号分隔部分
   (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]{0,61}[a-z0-9]\. ) *
   #  后缀
   (?: com|edu|gov|int|mil|net|org|biz|info|…|aero|[a-z][a-z] )
}xi;
```

它的使用方式与之前的例子相同（开头没有「^」，结尾没有「$」，也没有捕获型括号），所以，我们的替换不受限制。这样能得到更准确的$HttpUrl。

匹配模式（即使不设置）是不可更改的

qr/…/支持292页介绍的核心修饰符。regex对象一旦构建完成，对应的匹配模式就不能更改了，即使regex对象所在的m/…/有自己的修饰符也是如此。下面的代码就不正确：

```
my $WordRegex = qr/\b \w+ \b/;    #  这里忘了添加修饰符/x
   .
   .
   .
if ($text =~ m/^($WordRegex)/x) {
    print "found word at start of text: $1\n";
}
```

这里希望用/x修饰$WordRegex的匹配模式，但是这并不管用，因为在$WordRegex生成时修饰符（即使不设置）被锁定在qr/…/中。所以，修饰符必须在恰当的时候使用。

下面的代码则没有问题：

```
my $WordRegex = qr/\b \w+ \b/x;   #  没问题！
if ($text =~ m/^($WordRegex)/) {
   print "found word at start of text: $1\n";
}
```

现在来比较开始的代码和下面的代码：

```
my $WordRegex = '\b \w+ \b';      #  普通字符串
if ($text =~ m/^($WordRegex)/x) {
   print "found word at start of text: $1\n";
}
```

这段代码没问题，尽管`$WordRegex`生成时没有任何修饰符。因为`$WordRegex`是普通变量，保存普通的字符串，用作 `m/…/`的插值。因为各方面的原因，用字符串构建正则表达式比 regex 对象要麻烦得多，比如在这个例子中，必须记住`$WordRegex`必须和`/x`一起使用才行。

我们也可以只使用字符串解决这个问题，只需要在表达式中设定**模式修饰范围**（☞135）：

```
my $WordRegex = '(?x:\b \w+ \b)'; #  普通字符串
  ......
if ($text =~ m/^($WordRegex)/) {
   print "found word at start of text: $1\n";
}
```

此时，`m/…/`的正则文字插值之后，正则引擎接收到「`^((?x:\b·\w+·\b))`」，这就是我们期望的结果。

其实这就是生成 regex 对象的逻辑过程，只是 regex 对象对于每个模式修饰符，都明确定义了“on”或“off”的状态。用 **`qr/\b·\w+·\b/x`** 生成「`(?x-ism:\b·\w+·\b)`」。请注意模式修饰符的设定「`(?x-ism:…)`」，这里启用了`/x`，禁止了`/i`、`/s`和`/m`。也就是说，无论是否指定 `qr/…/`的修饰符，regex 对象总是“锁定”在某个模式下。

探究 regex 对象

Viewing Regex Objects

前面讨论了 regex 对象综合正则表达式和模式修饰符——例如「`(?x-ism:…)`」——的逻辑过程。如果在 Perl 期望接收字符串的地方使用 regex 对象，Perl 会把它转换为对应的文本表示方式，例如：

```
% perl -e 'print qr/\b \w+ \b/x, "\n"'
(?x-ism:\b \w+ \b)
```

这就是第304页的`$HttpUrl`的转换结果：

```
(?ix-sm:
  http:// (?ix-sm:
  #  一个或多个点号分隔部分
  (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]{0,61}[a-z0-9]\. ) *
  #  后缀
  (?: com|edu|gov|int|mil|net|org|biz|info|…|aero|[a-z][a-z] )
) \b # hostname
  (?:
      / [-a-z0-9-:\@&?=+,.!/~*'%\$]*        #  可能出现的path
          (?<![.,?!])                       #  不能以[.,?!]结尾
  )?
)
```

把regex对象转换为字符串的功能在调试时很有用。

用regex对象提高效率

Using Regex Objects for Efficiency

使用regex对象的主要原因之一是便于控制。为了提高效率，Perl会把正则表达式编译为内部形式。第6章简要介绍了正则表达式编译的一般知识，但是更复杂的Perl相关问题，包括regex对象之类，都在"正则表达式编译、/o修饰符、qr/…/和效率"（☞348）中详细讨论。

Match运算符

The Match Operator

基本的匹配操作：

```
$text =~ m/regex/
```

是Perl的正则表达式应用的核心。在Perl中，正则表达式匹配操作需要两个**运算元**，其一是目标字符串，其二是正则表达式，返回一个值。

匹配如何进行，返回什么值，取决于匹配的应用场合（☞294）及其他因素。match运算符非常方便——它可以用来测试某个正则表达式能否匹配一个字符串，或者从字符串中提取数据，甚至是与其他匹配运算符一起将字符串拆分为各个部分。虽然功能强大，这种便捷也增加了掌握的难度。需要关注的内容包括：

- 如何指定正则运算元。
- 如何指定匹配修饰符，以及它们的意义。
- 如何指定目标字符串。
- 匹配的伴随效应。
- 匹配的返回值。
- 能影响匹配的外部因素。

匹配的常见形式是：

```
StringOperand =~ RegexOperand
```

还有许多简便方式，值得注意的是，某些简便形式下，两个运算元都不是必须出现的。本节中我们会看到各种形式的例子。

Match 的正则运算元

Match's Regex Operand

正则运算元可以是正则文字或者 regex 对象（其实可以是字符串或者任意的表达式，但是这样做没什么好处）。如果使用正则文字，可以指定修饰符。

使用正则文字

正则运算元通常是 m/…/或者就是/…/内的正则文字。如果正则文字的分隔符是斜线或问号（以问号做分隔符的情况很特殊，稍后讨论）则可以省略开头的 m。为保持一致，我不管是否必要都使用 m。之前介绍过，如果使用 m，你可以使用自己的分隔符（☞291）。

使用正则文字时，可以结合第 292 页介绍的任何核心修饰符。匹配运算符还支持两个另外的运算符，/c 和/g，下面马上介绍。

使用 regex 对象

正则运算元也可以是 qr/…/生成的 regex 对象，例如：

```
my $regex = qr/regex/;
  ......
if ($text =~ $regex) {
  ......
```

可以在 m/…/中使用 regex 对象。特殊的情况是，如果“正则文字”中只包含一个 regex 对象的插值，那么它就完全等同于使用此 regex 对象。上面这个例子也可以写作：

```
if ($text =~ m/$regex/) {
  ......
```

这很方便，因为它看起来更熟悉，也容许我们对 regex 对象使用/g 修饰符（还可以使用 m/…/支持的其他的修饰符，但这在本例中没有意义，因为它们不能覆盖 regex 对象内锁定的模式修饰符☞304）。

默认的正则表达式

如果没有指定正则表达式，例如 m//（或者 m/$SomeVar/而变量$SomeVar 为空字符串或未定义），则 Perl 会使用此代码所在的**动态作用域范围内最近应用成功的正则表达式**。以前这样很有用，因为可以提高效率，后来因为 regex 对象的发展（☞303），已经没什么意义了。

?…?的特殊匹配

除了之前介绍的正则文字的各种分隔符，match 运算符还可以使用一种特殊的分隔符——问号。问号分隔符（例如 m?…?）提供的是相当生僻的功能，m?…?匹配成功之后，除非在同样的 package 中调用 reset 函数，否则不会再次匹配。按照 Perl Version 1 的手册上的说法，这"是有用的优化措施，用于在一组文件中查找某段信息的第一次出现"，但是不知何故，我在现代 Perl 中从未见过。

问号分隔符的特殊情况类似斜线分隔符，m 也不是必须出现的：?…?完全等价于 m?…?。

指定目标运算元

Specifying the Match Target Operand

常用来指定"搜索字符串"的做法是=~，例如**$text =~ m/…/**。请记住，=~既不是赋值运算符，也不是比较运算符，只是一个看来有趣的运算符，用来连接运算元（这个表示法改编自 awk）。

因为整个"***expr* =~ m/…/**"本身就是一个表达式，我们可以在任何容许出现表达式的地方使用，例如（以连线分隔）：

```
$text =~ m/…/;   #  这样做，大概是从伴随效应考虑的
---------------------------------
if ($text =~ m/…/) {
    #  如果匹配成功，执行这些代码
    ……

---------------------------------
  $result = ( $text =~ m/…/ );     #  把$result 置为$text 的匹配结果
  $result = $text =~ m/…/ ;        #  同上 =~ 的优先级高于 =
---------------------------------
  $copy = $text;                   #  把$text 拷贝到$copy...
  $copy              =~ m/…/;      #  ... 在$copy 上匹配
 ( $copy = $text )  =~ m/…/;       #  同上
```

默认的目标字符串

如果目标字符串是变量$_，则可以省略整个"**$_ =~**"。也就是说，默认的目标字符串就是$_。

```
$text =~ m/regex/;
```

的意思是，“把 regex 应用到`$text`中的文本，忽略返回值，获取伴随效应”。如果忘了写‘~’，结果就是：

```
$text = m/regex/;
```

它的意思是“对`$_`中的文本应用正则表达式，获取伴随效应，把返回的 true/false 值赋给`$text`”。也就是说，下面两者是等价的：

```
$text =        m/regex/;
$text = ($_ =~ m/regex/);
```

有时候使用默认目标字符串很方便，尤其是与其他默认情况的结构（许多结构都有默认值）结合时。下面的代码就很常见：

```
while (<>)
{
   if (m/…/) {
     ……

   } elsif (m/…/) {
     ……
```

总的来说，依赖默认运算元会增加无经验程序员阅读代码的难度。

颠倒 match 的意义

可以用`!~`来取代`=~`，对返回值进行逻辑非操作（马上会介绍这么做的返回值和伴随效应，但是对于`!~`，返回值就是 true 或者 false），下面三种办法是等价的：

```
if ($text !~ m/…/)
if (not $text =~ m/…/)
unless ($text =~ m/…/)
```

从我个人出发，我喜欢中间的办法。无论选用哪种办法，都会产生设置`$1`等的伴随效应。`!~`只是判断“如果不能匹配”的简便方式。

Match 运算符的不同用途

Different Uses of the Match Operator

可以从 match 运算符返回的 true/false 判断匹配是否成功，也可以从成功匹配中获取其他的信息，与其他 match 运算符结合起来。match 运算符的行为主要取决于它的应用场合(☞294)，以及是否使用了`/g`修饰符。

普通的"匹配与否"——scalar context，不使用/g

在 scalar context 中（例如 if 测试），match 运算符返回的就是 true/false：

```
if ($target =~  m/…/) {
    #  ... 匹配成功后的操作 ...
} else {
    #  ... 匹配失败后的操作 ...
}
```

也可以把结果赋值给一个 scalar 变量，然后检查

```
my $success = $target =~  m/…/;
  ......
if ($success) {
  ......
}
```

普通的"从字符串中提取数据"——list context，不使用/g

不使用/g 的 list context，是字符串中提取数据的常用做法。返回值是一个 list，每个元素是正则表达式中捕获型括号内的表达式捕获的内容。下面这个简单的例子用来从 69/8/31 中提取日期：

```
my ($year, $month, $day) = $date =~ m{ ^ (\d+) / (\d+) / (\d+)  $ }x;
```

匹配的 3 个数作为 3 个变量（当然还包括$1、$2 和$3 等）。每一组捕获型括号都对应到返回序列中的一个元素，空序列表示匹配失败。

有时候，某组捕获括号没有参与最终的成功匹配。例如，m/(this)|(that)/必然有一组括号不会参与匹配。这样的括号返回未定义的值 undef。如果匹配成功，又没有使用捕获型括号，在不使用/g 的 list context 中，会返回 list(1)。

List context 可以以各种方式指定，包括把结果赋值给一个数组，例如：

```
my @parts = $text =~ m/^(\d+)-(\d+)-(\d+)$/;
```

如果 match 的接收参数是 scalar 变量，请将匹配的应用场合指定为 list context，这样才能获得匹配的某些捕获内容，而不是表示匹配成功与否的 Boolean 值。比较这两个测试：

```
my ($word)   = $text =~ m/(\w+)/;
my $success = $text =~ m/(\w+)/;
```

第一个例子中，变量外的括号导致 my 函数为赋值指定 list context。第二个例子没有括号，所以应用场合为 scalar context，$success 只得到 true/false 值。

下面给出了一个更简单的做法：

```
if ( my ($year, $month, $day) = $date =~ m{^ (\d+) / (\d+) / (\d+) $}x ) {
    # 如果能够匹配，$year 等变量已经赋值
} else {
    # 如果不能匹配...
}
```

这次匹配发生在 list context 中（由“my (…) =”提供），所以序列中的变量会根据对应的 $1、$2 之类进行赋值。不过，匹配完成之后，因为整个组合是在 if 条件语句的 scalar context 中，Perl 把 list 转换为一个 scalar 变量。它接收的是 list 的长度，如果匹配不成功，长度为 0，如果不为 0，则表示匹配成功。

“提取所有匹配”——list context，使用/g

此结构的用处在于，它返回一个文本序列，每个元素对应捕获型括号匹配的文本（如果没有捕获型括号，就返回整个表达式匹配的文本），但上一节的例子只能针对一次匹配，而这种结构针对所有匹配。

下面这个简单的例子用来提取字符串中的所有整数：

```
my @nums = $text =~ m/\d+/g;
```

如果$text 包含 IP 地址‘64.156.215.240’，@num 会接收 4 个元素，‘64’、‘156’、‘215’、‘240’。与其他结构相结合，就能很方便地把 IP 地址转换为 8 位 16 进制数字，例如‘409cd7f0’，如果需要创建紧凑的 log 文件，这很方便：

```
my $hex_ip = join '', map { sprintf("%02x", $_) } $ip =~ m/\d+/g;
```

下面的代码可以把它转换回来：

```
my $ip = join '.', map { hex($_) } $hex_ip =~ m/../g
```

另一个例子是匹配一行中的所有浮点数：

```
my @nums = $text =~ m/\d+(?:\.\d+)?|\.\d+/g;
```

一定要使用非捕获型括号，因为捕获型括号会改变返回的结果。下面的例子说明了捕获型括号的价值：

```
my @Tags = $Html =~ m/<(\w+)/g;
```

@Tags 会保存$Html 中依次出现的各个 tag，这里假设每一个‘<’都有对应的‘>’。

下面的例子使用了多个捕获型括号：把 Unix 中邮箱联系人的 alias 文件的内容存放在一个字符串中，数据格式是：

```
alias Jeff      jfriedl@regex.info
alias Perlbug   perl5-porters@perl.org
alias Prez      president@whitehouse.gov
```

为了提取每一行中的昵称（alias）和完整地址，我们可以使用 **`m/^alias\s+(\S+)\s+(.+)/m`**（不使用`/g`）。在 list context 中，返回的序列包括两个元素，例如`('Jeff', 'jfriedl@regex.info')`。现在用`/g` 匹配所有这样的组合，得到的序列是：

```
( 'Jeff', 'jfriedl@regex.info', 'Perlbug',
'perl5-porters@perl.org', 'Prez', 'president@whitehouse.gov' )
```

如果这个序列恰好符合 key/value 的形式，我们可以直接把它存入一个关联数组（associative array）。

```
my %alias = $text =~ m/^alias\s+(\S+)\s+(.+)/mg;
```

返回之后，可以用`$alias{Jeff}`访问`'Jeff'`的完整地址。

迭代匹配：Scalar Context，使用/g

Iterative Matching: Scalar Context, with /g

scalar context 中，`m/…/g` 是个特殊的结构。和正常的 `m/…/` 一样，它只进行一次匹配，但是和 list context 中的 `m/…/g` 一样，它会检查之前匹配的发生位置。每次在 scalar context 中使用 `m/…/g`——例如在循环中，它会寻找“下一个”匹配。如果失败，就会重置“当前位置（current position）”，于是下一次应用从字符串的开头开始。

这里有个简单的例子：

```
$text = "WOW! This is a SILLY test.";
$text =~ m/\b([a -z]+\b)/g;
print "The first all-lowercase word: $1\n";
$text =~ m/\b([A-Z]+\b)/g;
print "The subsequent all-uppercase word: $1\n";
```

有两次匹配是在 scalar context 中使用`/g`进行的，结果为：

```
The first all-lowercase word: is
The subsequent all-uppercase word: SILLY
```

后两次匹配配合起来，前者把“当前位置”设置到匹配的小写字母单词之后，第二个读取这个位置，在后面寻找大写字母单词。对两个匹配来说，`/g` 都是必须的，这样匹配才能注意到“当前位置”，所以如果**任何一个**没有使用`/g`，第二行会指向‘`WOW`’。

scalar context 中的/g 匹配非常适合用作 while 循环的条件：

```
while ($ConfigData =~ m/^(\w+)=(.*)/mg) {
    my($key, $value) = ($1, $2);
     ......
}
```

最终会找到所有的匹配，但是 while 的循环体是在匹配之间（或者说，在每次匹配之后）执行。一旦某次匹配失败，结果就是 false，然后 while 循环结束。同样，一旦失败，/g 状态会重置，也就是循环结束之后的/g 匹配会从字符串的开头开始。

比较：

```
while ($text =~ m/(\d+)/) {   #  很危险!
   print "found: $1\n";
}
```

和

```
while ($text =~ m/(\d+)/g) {
    print "found: $1\n";
}
```

唯一的区别是/g，但是这区别不可小视。如果$text 包含之前那个 IP 地址，第二个程序输出我们期望的结果：

```
found: 64
found: 156
found: 215
found: 240
```

相反，第一个程序不断地打印“found: 64”，不会终止。不使用/g，就意味着“找到$text 中第一个「(\d+)」”，也就是‘64’，无论匹配多少次都是如此。添加/g 之后，它变成了“找到$text 中的下一个「(\d+)」”，依次找到各个数字。

“当前匹配位置”和 pos()函数

Perl 中每个字符串都有对应的“当前匹配位置（current match position）”，传动装置会从这里开始第一次匹配的尝试。这是字符串的属性之一，而与正则表达式无关。在字符串创建或者修改时，“当前匹配位置”会指向字符串的开头，但是如果/g 匹配成功，它就会指向本次匹配的结束位置。下一次对字符串应用/g 匹配时，匹配会从同样的“当前匹配位置”开始。

可以通过 pos(…) 函数得到目标字符串的“当前匹配位置”，例如：

```
my $ip = "64.156.215.240";
while ($ip =~ m/(\d+)/g) {
   printf "found '$1' ending at location %d\n", pos($ip);
}
```

结果是：

```
found '64' ending at location 2
found '156' ending at location 6
found '215' ending at location 10
found '240' ending at location 14
```

（记住，字符串的下标是从0开始的，所以“location 2”就是第3个字符之前的位置）。在 /g 匹配成功之后，$+[0]（@+的第一个元素☞302)就等于目标字符串中的pos。

pos() 函数的默认参数是match运算符的默认参数：变量$_。

预设定字符串的pos

pos() 的真正能力在于，我们可以通过它来指定正则引擎从什么位置开始匹配（当然是针对 /g 的下一次匹配）。我在Yahoo!的时候，要处理的Web服务器log文件的格式是，包含32字节的定长数据，然后是请求的页面，然后是其他信息。提取请求页面的办法是「^.{32}」，跳过开头的定长数据：

```
if ($logline =~ m/^.{32}(\S+)/) {
   $RequestedPage = $1;
}
```

这种硬办法不够美观，而且要强迫正则引擎处理前32个字节。如果我们亲自动手，代码会好看得多，效率也高得多：

```
pos($logline) = 32;      # 请求页的信息从第33个字符开始...
if ($logline =~ m/(\S+)/g) {
   $RequestedPage = $1;
}
```

这个程序好些，但还不够好。这个正则表达式从我们规定的位置**开始**，而在此之前不需要匹配，**这一点**与上个程序不同。如果因为某些原因，第32个字符不能由「\S」匹配，前面那个程序就会匹配失败，但是新程序因为没有锚定到字符串的特殊位置，会由传动装置启动驱动过程。也就是说，它会错误地返回一个「\S+」在字符串后面部分的匹配。幸好，在下一节我们会看到，这个问题很容易修复。

使用\G

元字符\G 的意思是“锚定到上一次匹配结束位置”。这正是上一节中我们希望解决的问题。

```
pos($logline) = 32;  # 设定“当前位置”从第 32 个字符开始，所以从此处开始匹配...
if ($logline =~ m/\G(\S+)/g)
{
   $RequestedPage = $1;
}
```

\G 告诉传动装置，“不要启动驱动过程，如果在此处匹配不能成功，就报告失败”。

前面几章曾经介绍过「\G」：第 3 章有简单介绍（☞130），更复杂的例子在第 5 章（☞212）。

请注意，在 Perl 中，只有「\G」出现在正则表达式开头，而且没有全局性多选结构的情况下，结果才是可预测的。第 6 章的优化 CSV 解析程序的例子中（☞271），正则表达式以「\G(?:^|,)…」开头。如果更严格的「^」能够匹配，就没必要检查「\G」，所以你可能会把它改为「(?:^|\G,)…」。不幸的是，在 Perl 中这样行不通，其结果不可预测（注 7）。

使用/gc 进行“Tag-team”匹配

正常情况下，m/…/g 匹配失败会把目标字符串的 pos 重置到字符串的开头，但给/g 添加/c 之后会造成一种特殊的效果：匹配失败**不会**重置目标字符串的 pos（/c 离不开/g，所以我一般使用/gc）。

m/…/gc 最常见的用法是与「\G」一起，创建“词法分析器”，把字符串解析为各个记号(token)。下例简要说明了如何解析$html 中的 HTML 代码：

```
while (not $html =~ m/\G\z/gc)    # 在全部检查完之前...
{
  if    ($html =~ m/\G( <[^>]+>    )/xgc){ print "TAG: $1\n"            }
  elsif ($html =~ m/\G( &\w+;      )/xgc){ print "NAMED ENTITY: $1\n"   }
  elsif ($html =~ m/\G( &\#\d+;    )/xgc){ print "NUMERIC ENTITY: $1\n" }
  elsif ($html =~ m/\G( [^<>&\n]+ )/xgc){ print "TEXT: $1\n"            }
  elsif ($html =~ m/\G  \n          /xgc){ print "NEWLINE\n"             }
  elsif ($html =~ m/\G( .          )/xgc){ print "ILLEGAL CHAR: $1\n"   }
  else {
      die "$0: oops, this shouldn't happen!";
  }
}
```

注 7：在大多数支持「\G」流派中这样都没有问题，即便如此，我一般也不推荐使用它们，因为把「\G」放在正则表达式开头带来的收益大于只在某些情况下测试「\G」的收益（☞246）。

每个正则表达式的粗体部分匹配一种类型的HTML结构。从当前位置开始，依次检查每一个正则表达式（使用/gc），但是只能在当前位置尝试匹配（因为使用了「\G」）。按照顺序依次检查各个正则表达式，直到找到能够匹配的结构为止，然后报告。之后把$html 的 pos指向下一个记号的开始，进入下一轮循环的搜索。

循环终止的条件是m/\G\z/gc能够匹配，即当前位置（「\G」）指向字符串的末尾（「\z」）。

有一点要注意，每轮循环**必须**有一个匹配。否则（而且我们不希望退出）就会进入无穷循环，因为$html 的 pos既不会变化也不会重置。对现在的程序来说，最终的 *else* 分支永远不会调用，但是如果我们希望修改这个程序（马上就会这么做），或许会引入错误，所以 *else* 分支是有必要保留的。对目前这个程序来说，如果接收预料之外的数据（例如'<>'），会在每次遇到预料之外的字符时，就发出一条警报。

另一点需要注意的是各表达式的检查顺序，例如把「\G(.)」作为最后的检查。也可以来看下面这个识别<script>代码的例子：

```
$html =~ m/\G ( <script[^>]*>.*?</script> )/xgcsi
```

哇，这里使用了5个修饰符！为了正常运行，我们必须把它放在对字符串进行第一次「<[^>]+>」的匹配**之前**。否则「<[^>]+>」会匹配开头的<script>标签，这个表达式就没法运行了。

第3章还介绍了关于/gc的更高级的例子（☞132）。

Pos相关问题总结

下面是match运算符与目标字符串的pos之间互相作用的总结：

匹配类型	尝试开始位置	匹配成功时的pop值	匹配失败时的pop设定
m/…/	字符串起始位置（忽略pos）	重置为undef	重置为undef
m/…/g	字符串的pos位置	匹配结束位置的偏移值	重置为undef
m/…/gc	字符串的pos位置	匹配结束位置的偏移值	不变

同样，只要修改了字符串，pos就会重置为undef（也就是初始值，指向字符串的起始位置）。

Match运算符与环境的关系

The Match Operator's Environmental Relations

下面几节将总结我们已经见到的，match运算符与Perl环境之间的互相影响。

match 运算符的伴随效应

通常，成功匹配的伴随效应比返回值更重要。事实上，在 void context 中使用 match 运算符（这样甚至不必检查返回值），只是为了获取伴随效应（这种情况类似 scalar context）。下面总结了成功匹配的伴随效应：

- 匹配之后会设置`$1` 和`@+`之类变量，供当前语法块内其他代码使用（☞299）。
- 设置**默认**正则表达式，供当前语法块内其他代码使用（☞308）。
- 如果 `m?…?`能够匹配，它（也就是 `m?…?`运算符）会被标记为无法继续匹配，至少在同样的 package 中，不调用 `reset` 就无法继续匹配（☞308）。

当然，这些伴随效应只能在匹配成功时发生，不成功的匹配不会影响它们。相反，下面的伴随效应在**任何**匹配中都会发生：

- 目标字符串的 `pos` 会指定或者重置（☞313）。
- 如果使用了`/o`,正则表达式会与这个运算符"融为一体(fuse)",不会重新求值(evaluate，☞352)。

match 运算符的外部影响

match 运算符的行为会受到运算元和修饰符的影响。下面总结了影响 match 运算符的外部因素：

应用场合 context

match 运算符的应用场合（scalar、list，或者 void）对匹配的进行、返回值和伴随效应有重要影响。

`pos(…)`

目标字符串的 `pos`（由前一次匹配显式或隐式设定）表示下一次**`/g`** 匹配应该开始的位置，同时也是「`\G`」匹配的位置。

默认表达式

如果提供的正则表达式为空，就使用默认的表达式（☞308）。

`study`

对匹配的内容或返回值没有任何影响，但如果对目标字符串调用此函数，匹配所花的时间更少（也可能更多）。参考"Study 函数"（☞359）。

`m?…?`和 `reset`

`m?…?`运算符有一个看不见的"已/未匹配"状态，在使用 `m?…?`匹配或者 `reset` 时设定（☞308）。

在 context 中思考（不要忘记 context）

在 match 运算符讲解结束之前，我要提几个问题。尤其是，在 while、if 和 foreach 控制结构中发生变化时，确实需要保持头脑清醒。请问，运行下面的程序会得到什么结果？

```
while  ("Larry Curly Moe" =~ m/\w+/g) {
  print "WHILE stooge is $&.\n";
}
print "\n";

if ("Larry Curly Moe" =~ m/\w+/g) {
  print "IF stooge is $&.\n";
}
print "\n";

foreach ("Larry Curly Moe" =~ m/\w+/g) {
  print "FOREACH stooge is $&.\n";
}
```

这有点儿难度，◆ 请翻到下页查看答案。

Substitution 运算符

The Substitution Operator

Perl 的 substitution 运算符 s/…/…/不但能够匹配，还能够替换匹配的文字。通常的形式是：

```
$text =~ s/regex/replacement/modifiers
```

简单来说，regex 匹配的文本会替换为 replacement 的值。如果使用了/g，这个正则表达式会重复应用到文本中进行匹配，每次匹配的内容都会被替换。

与 match 操作一样，如果目标字符串在变量$_中，目标运算元和=~都不是必须的。match 运算符可以省略 m，而 substitution 不能省略 s。

我们已经看到，match 运算符是非常复杂的——它的工作原理，它的返回值，都取决于它所在的应用场合，目标字符串的 pos，以及使用的修饰符。相反，substitution 运算符很简单：它返回的信息是不变的（表示替换的次数），影响它的修饰符也很好理解。

你可以使用第 292 页介绍的所有核心修饰符，但是 substituion 运算符还支持另外两个修饰符，/g，以及马上将要介绍的/e。

运算元 replacement

The Replacement Operand

在普通 `s/…/…/`中，replacement 紧跟在 regex 之后；`m/…/`使用两个分隔符，而这里要使用 3 个。如果正则表达式使用对称的分隔符（例如`<…>`），则 replacement 有自己的一对分隔符（这样总共就有 4 个分隔符）。举例来说，`s{…}{…}`和 `s[…]/…/`和 `s<…>'…'`都是合法的。这种情况下，两对分隔符可以用空白字符分隔，如果使用了空白字符，还可以添加注释。对称的分隔符通常在`/x` 或`/e` 中使用。

```
$text =~ s{
    ...复杂的表达式，大量的注释，以及...
} {
    ...对一段 Perl 代码求值，把结果作为 replacement...
}ex;
```

请注意区分 regex 和 replacement。regex 会按照正则表达式的方式来解析，有自己的分隔符（☞291）。replacement 则会当作普通的双引号字符串来解析和求值（evaluate）。求值会在匹配之后进行（如果使用了`/g`，每次匹配之后都会求值），所以`$1` 之类的变量能够指向对应的匹配内容。

在下面两种情况下，replacement 不会按照双引号字符串来解析：

- replacement 的分隔符是单引号，此时作为单引号字符串，不会进行变量插值。
- 使用了`/e` 修饰符（下一节讨论），replacement 会作为一小段 Perl 代码而不是双引号字符串。这一小段 Perl 代码会在每次匹配之后执行，结果作为 replacement。

/e 修饰符

The /e Modifier

`/e` 修饰符会把 replacement 作为一段 Perl 代码来进行求值，这就类似 **`eval {…}`**。代码装载时，首先会检查这段代码的语法，确保没有错误，但是每次匹配之后都会对代码重新求值。每次匹配之后，replacement 都会在 scalar context 中重新求值，结果作为 replacement。下面有个简单的例子：

```
$text =~ s/-time-/localtime/ge;
```

在 scalar context 中，它会用 Perl 的 `localtime` 函数的结果（也就是返回表示当前时间的文本，例如“`Mon Sep 25 18:36:51 2006`”）替换「`-time-`」。

因为求值是每次匹配之后进行的，我们可以通过`$1` 等变量引用匹配的内容。例如，URL 中

测验答案

◆ 318 页测验的答案

318 页代码的结果：

```
WHILE stooge is Larry.
WHILE stooge is Curly.
WHILE stooge is Moe.

IF stooge is Larry.

FOREACH stooge is Moe.
FOREACH stooge is Moe.
FOREACH stooge is Moe.
```

请注意，如果 foreach 循环中的 print 引用了$_而不是$&，结果就会与 while 的一样。在这个 foreach 中，m/…/g 返回的('Larry', 'Curly', 'Moe')并没有使用。相反倒是使用了伴随效应中的$&，这表示程序有错误，因为 list context 的伴随效应，m/…/g 并不常用。

不容许出现的特殊字符，可以编码为百分号“%”加两位十六进制数的形式。为了编码所有这种字符，可以这样：

```
$url =~ s/([^a-zA-Z0-9])/sprintf('%%%02x',ord($1))/ge;
```

下面的程序可以用来解码：

```
$url =~ s/%([0-9a-f][0-9a-f])/pack("C", hex($1))/ige;
```

简单地说，**sprintf('%%%02x', ord(*character*))**把字符转换为对应的 URL 编码，而**pack("C", *value*)**的作用相反，请参考你常用的 Perl 文档获取更多信息。

多次使用/e

通常情况下，对单个运算符多次使用同一修饰符没有特殊意义（只有让读者更困惑），但是重复/e 修饰符却会改变 replacement 的替换过程。正常情况下，replacement 会进行一次求值，但是如果‘e’的数目多于 1 个，则 Perl 又会对求值的结果进行求值，如此一直进行下去，求值的次数与‘e’的数目一样多。或许它的主要价值是用作比较 Perl 代码复杂性的测试。

不过此功能并非完全无用。如果需要手动进行变量插值（例如从配置文件读入字符串）。也就是说，有一个字符串‘…$var…’，我们希望把‘$var’替换为$var 的值。

简单的办法是：

```
$data =~ s/(\$[a-zA-Z_]\w*)/$1/eeg;
```

如果不使用/e，则会替换匹配的‘$var’自身，这没什么用。使用一个/e，会对$1重新求值，得到‘$var’，这样也没什么意义，同样是用匹配的文本替换自身。但是如果使用两个/e，则‘$var’会重新求值，得到内容，这样就模拟了变量插值。

应用场合与返回值

Context and Return Value

根据 context 和/g 的不同组合，match 运算符会返回不同的值。不过，substitution 运算符没有这么复杂——它返回的要么是替换发生的次数，要么是空字符串，表示没有发生任何替换。

为使用方便，返回值为 Boolean 时（例如在 if 条件语句中），只要发生了替换，返回值就为 true，false 表示没发生替换。

Split 运算符

The Split Operator

功能多样的 split 运算符（在不那么严格的时候，人们通常称其为**函数**）常被视为 list context 中 m/…/g （☞311）的对立物。后者返回表达式匹配的文本，而 split 返回由表达式匹配的文本**分隔**的文本。把 **`$text =~ m/:/g`** 应用到 **`'IO.SYS:225558:95-10-03:-a-sh:optional'`** 中，返回四个元素的 list：

```
(':', ':', ':', ':' )
```

这似乎没什么用，但是 **`split(/:/, $text)`** 返回 5 个元素的 list：

```
('IO.SYS', '225558', '95-10-03', '-a-sh', 'optional' )
```

两个例子都告诉我们，「:」匹配了 4 次。使用 split，这 4 次匹配把目标字符串的副本分隔为 5 段，返回包含 5 个字符串的 list。

这个例子只用单个字符分隔目标字符串，其实我们可以使用任何正则表达式：

```
@Paragraphs = split(m/\s*<p>\s*/i, $html);
```

它会按照<p>或者<P>（两边可能有空白字符）把$html 中的 HTML 代码分隔开来。你甚至可以按位置分隔：

```
@Lines = split(m/^/m, $lines);
```

把字符串按行切分。

对最简单形式的数据来说，split 非常有用也很容易理解。不过，因为存在许多参数、特

殊情况和特殊情形，事情会变得复杂。在深入这些细节之前，先给出两个特别有用的例子：

- 特殊的 match 运算符`//`，会把目标字符串切分为单个字符，也就是说，**`split(//, "short test")`**得到 10 个元素的 list：**`("s", "h", "o", ..., "s", "t")`**。
- 特殊的 match 运算符"·"（包含单个空格的普通字符串）把目标字符串按照空白字符切分，等于使用 `m/\s+/`，只是会忽略开头和结尾的空白字符。这样，**`split("·", "···a·short···test···")`** 得到三个字符串：'a'、'short' 和 'test'。

稍后讨论各种特殊情况，我们先来看基础的部分。

Split 基础知识

Basic Split

`split` 运算符看起来像函数，它接收 3 个参数：

```
split(match operand, target string, chunk-limit operand)
```

括号是可选的，未提供的运算元会设置为默认值（本节稍后讨论）。

`split` 总是在 list context 中使用，常用的模式包括：

```
($var1, $var2, $var3, …) = split(…);
-------------------------------------
@array = split(…);
-------------------------------------
for my $item (split(…)) {
    ......
}
```

match 运算元

运算元 match 有几种特殊情况，不过它通常等价于 match 操作中的 `regex` 运算元。也就是说，你可以使用`/…/`和`m{…}`之类的形式，它可以是 regex 对象，或者任何能够求值为字符串的表达式。它只支持第 292 页介绍的核心修饰符。

如果继续要用括号来分组，请务必使用非捕获型括号「`(?:…)`」。我们稍后将会看到，在 split 中使用捕获型括号会触发极特殊的功能。

target string 运算元

target string 只用于检测，绝不会被修改。如果没有设定，默认使用`$_`。

chunk-limit 运算元

chunk-limit 运算元的主要功能是设定 split 切分字符串的数目上限。对上面的例子中同样的目标字符串调用 **split(/:/, $text, 3)**得到：

```
( 'IO. SYS', '225558', '95-10-03:-a-sh:optional' )
```

这告诉我们，/:/匹配两次之后 split 会终止，产生所需的 3 段。它可能可以匹配更多的次数，但这里不需要，因为存在段的数目限制。设定的数目将作为上限，所以最多只能返回规定数目的元素（除非正则表达式包含捕获型括号，后文会论及）。得到的元素数目可能少于上限，如果得到的分段少于规定的数目，也不会有多余的内容来“填充”。对于示例所用的数据，**split(/:/, $text, 99)**返回的 list 只有 5 个元素。不过，**split(/:/, $text)** 和 **split (/:/,$text, 99)**有重要的区别，这里暂时还看不出来——请记住这一点，稍后我们会仔细讲解。

记住，*chunk*-limit 运算元指向的是各匹配之间的分段，而不是匹配的数目本身。否则，前面那个上限为 3 的例子就应该得到这个：

```
('IO.SYS', '225558 ', '95-10-03', '-a-sh:optional' )
```

这不是程序运行的结果。

这里谈谈效率：假设只希望取开头的几个元素，例如：

```
($filename, $size, $date) = split(/:/, $text);
```

为了提高效率，在需要的元素赋值之后，Perl 会停止 split 操作。它会自动把 chunk-limit 设置为 list 的元素个数+1。

深入 split

从我们接触过的例子来看，split 很容易使用，但有三个特殊的问题增加了它实践起来的复杂程度：

- 返回空元素。
- 特殊的 regex 运算符。
- 包含捕获型括号的 regex。

下面分别讨论。

返回空元素

Returning Empty Elements

split 的基本功能是返回由各个匹配分隔的分本，但有的时候，返回的文本是空字符串（长度为 0 的字符串，例如""），比如：

```
@nums = split(m/:/, "12:34::78")
```

它返回：

```
("12","34","","78")
```

正则表达式「:」匹配了 3 次，所以应当返回 4 个元素。第 3 个元素为空，表示正则表达式在一行中匹配了两次，它们之间没有文本。

结尾的空元素

通常情况下，结尾的空元素不会返回，例如：

```
@nums = split(m/:/, "12:34::78:::");
```

同样会返回 4 个元素：

```
("12", "34", "", "78")
```

即使这个正则表达式在字符串的末尾能匹配更多的次数，结果也没有变化。在默认情况下，split 不会返回 list 末尾的空元素。不过，我们可以通过设定 chunk-limit 运算元，让 split 返回所有的末尾元素。

chunk-limit 运算元的次要职能

chunk-limit 的主要用途是设定分段数目的上限，任何不等于 0 的 chunk-limit 都会保留末尾的空元素（chunk-limit 设置为零等价于不设置 chunk-limit）。如果你不希望限制返回的 chunk 的数目，但是希望保留末尾的空元素，只需要设置一个非常大的限制即可。或者更好的办法是，设置为-1，因为 chunk-limit 为负数表示一个足够大的上限：**split(/:/, $text, -1)** 会返回所有的元素，包括末尾的空元素。

另一个极端是，如果你不希望保留任何空元素，可以在 split 之前使用 **grep{length}**。使用 grep 之后，只会返回长度不为 0 的元素（也就是说，非空元素）。

```
my @NonEmpty = grep { length } split(/:/, $text);
```

字符串末尾的特殊匹配

在字符串开头的匹配会产生一个空元素：

```
@nums = split(m/:/, ":12:34::78")
```

@num 的值为：

```
("", "12", "34", "", "78")
```

开始的空元素表明，正则表达式在字符串的开头能够匹配。不过也有例外，如果一个正则表达式没有匹配任何文本，如果它在字符串的开头或者结尾匹配，那么开头和/或结尾的空元素将不会生成。来看个简单的例子：**split(/\b/, "a simple test")**，它能够匹配其中的 6 个位置 '‸a‸·‸simple‸·‸test‸'。即使它能匹配 6 次，也不会返回 7 个元素，而是 5 个元素：**("a", "·", "simple", "·", "test")**。其实，这种特殊情况我们已经见过，即第 321 页的**@Lines = split(m/^/m, $lines)**。

Split 中的特殊 Regex 运算元

Split's Special Regex Operands

split 的 match 运算元通常就是正则文字或者 regex 对象，这与 match 运算符的情况一样，不过也有例外：

- regex 为空的意思不是"使用当前的默认正则表达式"，而是把目标字符串分割为字符。在刚开始讨论 split 的时候我们见过这个例子，**split(//, "short test")**返回 10 个元素的 list：**("s", "h", "o", …, "s", "t")**。
- 如果 match 运算元是由单个空格构成的字符串（而不是正则表达式），则是另一种特殊情况。它基本等同与/\s+/，只是会忽略开头的空白字符。这主要是为了模拟 awk 对输入进行的默认的输入-记录-分隔（input-record-separator）操作，尽管对普通情况来说它也很有用。

  如果希望保留开头的空白字符，可以直接使用 m/\s+/。如果希望保留末尾的空白字符，只需要把 chunk-limit 设置为-1。
- 如果没有设置 regex 运算元，则默认使用一个空格符（上面提到的特殊情况）。这样，不带任何运算元的 split 就等价于 **split('·', $_, 0)**。
- 如果 regex 为「^」，会自动使用修饰符/m（增强型行锚点匹配模式）。（因为某些原因，「$」则不行）。因为明确使用 m/^/**m** 非常容易，我推荐这种更清晰的写法。m/^/m 是把包含多行文本的字符串按行切分的简便方法。

Split 不产生伴随效应

请注意，split 的 match 运算元看起来很像普通的 match 运算符，但是它没有任何伴随效应。split 中的正则表达式不会影响到之后的 match 或是 substitution 操作所用的默认正则表达

式，也不会设置$&、$'、$1 之类的变量。split 在伴随效应上完全独立于程序的其他部分（注8）。

Split 中带捕获型括号的 match 运算元

Split's Match Operand with Capturing Parentheses

捕获型括号会从整体上改变 split。一旦使用了捕获型括号，返回的 list 中会多出些独立的元素，它们对应于括号捕获的元素。也就是说，split 没有返回的部分或全部的文本，会包含在返回的 list 中。

例如，在处理 HTML 时，对下面的文本调用 split(/(<[^>]*>)/)：

```
…·and·<B>very·<FONT·color=red>very</FONT>·much</B>·effort…
```

返回：

```
( '... ·and·', '<B>','very·', '<FONT·color=red>',
'very', '</FONT>', '·much', '</B>', '·effort...' )
```

如果去掉捕获型括号，split(/<[^>]*>/)返回：

```
( '... ·and·', 'very·', 'very', '·much', '·effort...' )
```

多出来的元素不受分段上限的限制（chunk-limit 限制原来字符串切分之后的分段数目，而不是返回元素的数目）。

如果包含多个捕获型括号，每次匹配之后，list 会多出多个元素。如果某个捕获型括号没有参与匹配，对应的元素为 undef。

巧用 Perl 的专有特性

Fun with Perl Enhancements

最先由 Perl 提供的许多正则表达式概念，现在其他语言也提供了。包括非捕获型括号、环视（以及之后的逆序环视）、宽松排列模式（其实是大多数模式，实际上还包括配套的「\A」、「\z」和「\Z」）、固化分组、「\G」和条件判断结构。这些概念不再是 Perl 独有的，所以我把它们挪到本书的通用部分。

不过 Perl 者也没有停止创新，所以现在还有些重要的概念只有 Perl 提供。其中最有意思的是在匹配尝试中执行任意代码的功能。长期起来，Perl 的特点之一就是正则表达式与代码的紧密集成，但是此特性把集成提升到了新的高度。

注8：其实确实存在一种伴随效应，它在若干年前就被摒弃（deprecate）了，但是没有从语言中去除。如果在 scalar context 或者 void context 下使用 split，它会把结果写入@_变量（它也是用来传递函数参数的变量，所以万万不要在这两种应用场合下使用 split）。在这两种情况下使用 split，如果设置了 **use warning** 或是命令行参数-w，都会看到警报。

我们先来简单看看这个特性和目前 Perl 独有的其他特性，然后详细讲解。

动态正则结构「`(??{ perl code })`」

应用正则表达式时，每次遇到表达式中的这个结构，就会执行其中的 Perl 代码。执行的结果（或者是 regex 对象，或者是解释为正则表达式的字符串）会作为当前匹配的一部分即刻被应用。

「`^(\d+)(??{"X{$1}"})$`」中的动态正则结构以下画线标注，这个正则表达式匹配的行开头是一个数，然后是字符‘X’必须重现对应的次数，直到行末尾。

它能匹配‘3XXX’和‘12XXXXXXXXXXXX’，但不能匹配‘3X’或‘7XXXX’。

仔细看‘3XXX’就会发现，开头的「`(\d+)`」匹配‘3XXX’，把`$1`设为‘3’。之后正则引擎遇到动态正则结构，执行“`X{$1}`”，得到‘`X{3}`’，解释得到的「`X{3}`」作为当前正则表达式的一部分（匹配‘3XXX’），末尾的「`$`」匹配‘3XXX’，得到整体匹配。

下面我们会看到，匹配任意深度的嵌套结构时，动态正则结构尤其有用。

嵌套代码结构「`(?{ arbitrary perl code})`」

与动态正则结构一样，在正则表达式的应用过程中，遇到此结构也会执行其中的 Perl 代码，但是这个结构更为通用，因为代码不需要返回任何特定的值。通常也不会用到返回值（不过如果表达式之后的部分需要，可以通过变量`$^R`得到☞302）。

有一种情况会用到这段代码的执行结果：如果内嵌的代码结构用作「`(?` *if then*`|`*else*`)`」中的 *if* 条件（☞140）。此时，结果会解释为布尔值，根据它来决定执行 *then* 还是 *else* 分支。

内嵌代码可以干许多事情，相当有用的就是调试。下面这段程序会在每次应用正则表达式时显示一条信息，内嵌的代码结构用下画线标注：

```
"hava a nice day" =~ m{
    (?{ print "Starting match.\n"})
    \b(?: the | an | a )\b
}x;
```

测试中，正则表达式只匹配 1 次，但是信息会显示 6 次，说明传动装置在第 6 次尝试之前已经在 5 个位置应用了正则表达式，第 6 次可以完全匹配。

正则文字重载

正则文字重载能够让程序员先自行处理正则文字，再将它们交给正则引擎。它可以用来为 Perl 的正则流派扩展新的功能。例如，Perl 没有提供单独的单词起始和结束分隔符（只有「\b」），不过你可能希望使用\<和\>，让 Perl 能够识别这些结构。

正则重载有些重要的限制，严格制约了它的用途。在讲解\<与\>的例子时我们会看到这一点。

如果正则表达式中内嵌了 Perl 代码（无论是动态正则结构还是内嵌代码结构），最好是只使用全局变量，除非你明白关于 338 页讲解的 my 变量的重要知识。关于 my 变量的讨论，请参阅第 338 页。

用动态正则表达式结构匹配嵌套结构

Using a Dynamic Regex to Match Nested Pairs

动态正则表达式的主要用途之一是匹配任意深度的嵌套结构（长久以来人们认为正则表达式对此无能为力）。匹配任意深度的嵌套括号是个重要的例子。为了说明白动态正则如何解决这个问题，我们首先必须知道传统结构为什么不能解决这个问题。

匹配括号文本的简单表达式是「\(([^()])*\)」。在外层括号内不容许出现括号，所以不能容许嵌套（也就是，只容许深度为 0 的嵌套）。用 regex 对象来表示就是：

```
my $Level0 = qr/ \( ( [^()] )* \) /x;      # 括号内文本
  ......
if ($text =~ m/\b( \w+$Level0 )/x) {
  print "found function call: $1\n";
}
```

这能够匹配“substr($str, 0, 3)”，但不能配“substr($str, 0, (3+2))”，因为它包含嵌套的括号。现在修改正则表达式来处理它，也就是需要能够处理深度为 1 的嵌套。

容许深度为 1 的嵌套意味着，外部的括号里头可以出现括号。所以，我们需要修改匹配外层括号内文本的表达式「[^()]」，添加一个子表达式匹配内层括号里的文本。我们可以这样，$Level0 保存这样一个正则表达式，再从此往上叠加：

```
my $Level0 = qr/ \( ( [^()]          )* \) /x;    # 括号内文本
my $Level1 = qr/ \( ( [^()]| $Level0 )* \) /x;    # 1层嵌套
```

这里的$Level0 与之前的相同，新出现了$Level1，匹配对应深度的括号，加上$Level0，就得到深度为 1 的嵌套。

为了增加嵌套的深度，我们可以用同样的方法，通过$Level1（仍然使用$Level0）得到$Level2：

```
my $Level0 = qr/ \( ( [^()]           )* \) /x;    # 括号内文本
my $Level1 = qr/ \( ( [^()] | $Level0 )* \) /x;    # 1层嵌套
my $Level2 = qr/ \( ( [^()] | $Level1 )* \) /x;    # 2层嵌套
```

继续下去就是：

```
my $Level3 = qr/ \( ( [^()] ; $Level2 )* \) /x;    # 3层嵌套
my $Level4 = qr/ \( ( [^()] ; $Level3 )* \) /x;    # 4层嵌套
my $Level5 = qr/ \( ( [^()] ; $Level4 )* \) /x;    # 5层嵌套
```

图 7-1 说明了开始几层的情况：

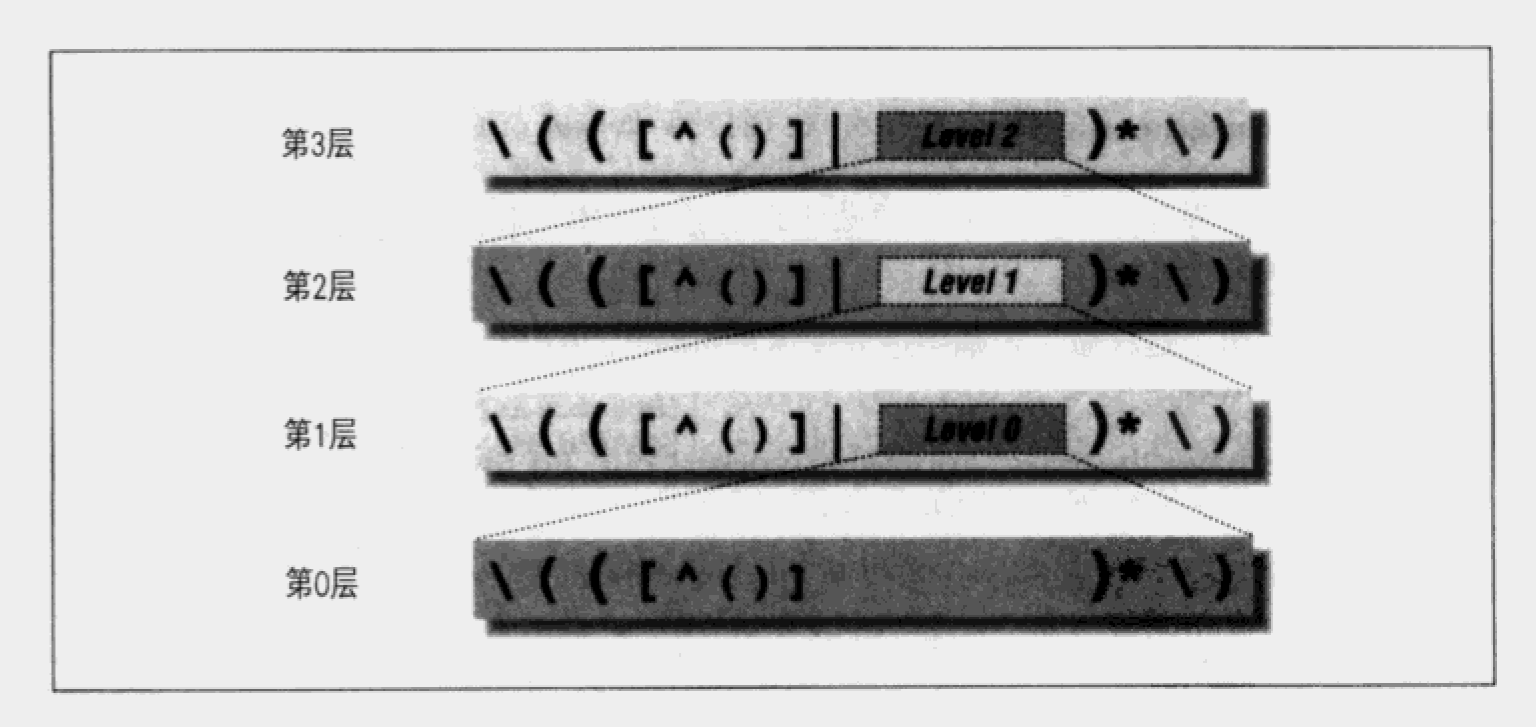


图 7-1：层数较少的嵌套

把这些层级加起来的结果很复杂，下面是$Level13：

```
\(([^()]|\(([^()]|\(([^()]|\(([^()])*\))*\))*\))*\)
```

这相当难看。

幸运的是，我们不需要直接解释它（那是正则引擎的工作）。使用$Level变量很容易处理，但问题是，嵌套的深度是由$Level变量的数目决定的。这种办法不够灵活（用墨非定律来说就是，如果程序能处理深度为X的嵌套，则遇到的数据的嵌套深度必定会是X+1）。

幸运的是，动态正则可以应付任意深度的嵌套。你只需要想明白，除第一个之外，每个$Level变量的构建方式都是相同的：需要增加一级嵌套深度时，只需要包含上一级的$Level变量即可。但如果$Level变量都是相同的，它就同样能包含更深级别的$Level。事实上，它还可以包括自身。如果在匹配更深层的嵌套时它可以用某种方式包含自身，就能递归地处理任意深度的嵌套。

这就是动态正则的威力所在。如果我们创建一个regex对象——比如$Level变量，就可以在动态正则中引用它（动态正则结构可以包含任意的Perl代码，只要结果能被解释为正则表达式，返回已存在的regex对象的Perl代码当然符合要求）。如果我们能把$Level之类的regex对象放入$LevelN，就可以用「(??{ $LevelN })」来引用它：

```
my $LevelN;              # 必须首先声明，下面才能使用
$LevelN = qr/ \(( [^()] | (??{ $LevelN }) )* \) /x;
```

它就能匹配任意深度的嵌套括号，用法同之前的$Level0：

```
if ($text =~ m/\b( \w+$LevelN )/x) {
   print "found function call: $1\n";
}
```

哈！想明白其中的道理可不是件容易的事情，不过一旦用过，就会发现这个工具的价值。

现在我们已经有了基本的办法，我希望做些修改提高效率。我会替换捕获型括号为固化分组（这里既不需要捕获文本，也不需要回溯），之后可以把「[^()]」改为「[^()]+」提高效率。（不要在固化分组中这样做，否则会造成无休止匹配☞226）。

最后，我希望把「\(」和「\)」移动到动态正则表达式两端。这样，在确实需要用到之前，引擎不会直接调用动态正则结构。下面是修改之后的版本：

```
$LevelN = qr/ (?> [^()]+ | \( (??{ $LevelN }) \) )* /x;
```

因为它不包含外部的「\(…\)」，调用$LevelN时必须手动添加。

这样一来，表达式就十分灵活，可以在任何**可能出现**嵌套括号的地方使用，而不仅仅是**出现了**嵌套括号的地方：

```
if ($text =~ m/\b( \w+ \( $LevelN \) )/x) {
    print "found function call: $1\n";
  }
  ----------------------------------
  if (not $text =~ m/^ $LevelN $/x) {
    print "mismatched parentheses!\n";
  }
```

第 343 页还有一个关于`$LevelN`的例子。

使用内嵌代码结构

Using the Embedded-Code Construct

内嵌代码结构很适合调试正则表达式，以及积累正在进行的匹配的信息。下面几页详细给出了一组例子，最终得到模拟 POSIX 匹配的方法。讲解的过程可能比真正的答案更有意思（除非你只需要 POSIX 的匹配语意），因为在讲解中我们会收获有用的技巧和启发。

先从简单的正则表达式调试技巧开始。

用内嵌代码显示匹配进行信息

这段程序：

```
"abcdefgh" =~ m{
  (?{ print "starting match at [$`|$']\n" })
  (?:d|e|f)
}x;
```

的结果是：

```
starting match at [|abcdefgh]
starting match at [a|bcdefgh]
starting match at [ab|cdefgh]
starting match at [abc|defgh]
```

正则表达式的开头就是内嵌代码结构，所以只要正则表达式开始新一轮匹配，就会执行：

```
print "starting match at [$`|$']\n"
```

它用变量`` $` ``和`$'`（☞300）（注 9）表示目标字符串，用‘|’标记当前的匹配位置（在这里就是匹配开始的位置）。从结果中我们可以知道，传动装置（☞148）进行了 4 次应用，才匹配成功。

注 9：通常，我不推荐使用特殊的匹配变量`` $` ``、`$&`和`$'`，它们会降低整个程序的效率（☞356），但是它们对临时调试非常有用。

事实上，如果我们添加：

```
(?{ print "matched at [$`<$&>$']\n" })
```

在正则表达式末尾，则结果是：

```
matched at [abc<d>efgh]
```

现在来看下面的程序，除了“主”正则表达式是「[def]」而不是「(?:d|e|f)」之外，其他部分与开头的例子是一样的：

```
"abcdefgh" =~ m{
  (?{ print "starting match at [$`|$']\n" })
  [def]
}x;
```

从理论上说，结果应该是一样的，实际情况却是：

```
starting match at [abc|defgh]
```

为什么呢？Perl 足够聪明，对这个以「[def]」开头的正则表达式进行开头字符/字符组/字串识别优化（☞247），这样传动装置就能略过那些它认为必然会失败的尝试。结果是忽略了其他所有尝试，只进行了可能导致匹配的尝试，我们可以通过内嵌代码结构观察到这种现象。

> **panic: top_env**
>
> 使用内嵌代码或是动态正则表达式时，如果程序忽然终止，给出这样的信息：
>
> `panic: top_env`
>
> 很可能是因为正则表达式的代码部分存在语法错误。Perl 不能识别某些错误的语法，结果就是这条信息。解决的办法就是修正语法错误。

用内嵌代码显示所有匹配

Perl 使用的是传统型 NFA 引擎，所以一旦找到匹配就会停下来，即使还存在其他的匹配也是如此。如果巧妙地使用内嵌代码，我们能够让 Perl 显示**所有的**匹配。我们仍然以 177 页的‘onself’为例来说明。

```
"oneselfsufficient" =~ m{
   one(self)?(selfsufficient)?
(?{print "matched at [$`<$&>$']\n" })
}x;
```

结果如我们所料：

```
matched at [<oneself>sufficient]
```

表示 ‘oneselfsufficient’ 已经被正则表达式匹配。

重要的是认识到，结果中的 “matched” 的部分并不是所有 “能够匹配” 的文本，只是到目前获得的匹配。在这个例子中谈论其中的区别意义不大，因为内嵌代码结构位于正则表达式的最后。我们知道，内嵌代码结构完成时，整个正则表达式的所有匹配尝试都已结束，实际匹配的结果就是如此。

不过，在内嵌代码结构之后添加「(?!)」的情况如何呢？「(?!)」是否定型顺序环视，它必然会失败。如果它在内嵌代码执行之后生效（也就是在 “matched” 信息打印之后），就会强迫引擎回溯，查找新的匹配。每次输出 “matched” 信息之后，「(?!)」都会强迫引擎回溯，最终试遍所有的可能。

```
matched at [<oneself>sufficient]
matched at [<oneselfsufficient>]
matched at [<one>selfsufficient]
```

我们所做的修改确保正则表达式必然不能完整匹配，但是这样做却能让引擎报告显示所有可能的匹配。如果不使用「(?!)」，Perl 只会返回第一个匹配，使用「(?!)」则可以见到其他可能。

了解了这一点之后，来看看下面的代码：

```
"123" =~ m{
   \d+
   (?{ print "matched at [$`<$&>$']\n" })
   (?!)
}x;
```

结果是：

```
matched at [<123>]
matched at [<12>3]
matched at [<1>23]
matched at [1<23>]
matched at [1<2>3]
matched at [12<3>]
```

前三行是我们能够想象的，但如果不仔细动动脑筋，可能没法理解后三行。「(?!)」强迫进行的回溯对应第二行和第三行。在开始位置的尝试失败之后，传动装置会启动驱动过程，从第二个字符开始（第 4 章对此有详细介绍）。第四行和第五行对应第二轮尝试，最后一行对应第三轮。

所以，添加(?!)之后确实能显示出**所有**可能的匹配，而不是从某个特定位置开始的所有匹配。不过，有时候只需要从特定位置开始的所有匹配，下面我们将会看到。

寻找最长匹配

如果我们不希望找到所有匹配，而是希望找到并保存最长的匹配，应该如何做呢？我们可以用一个变量来保存“到目前为止”最长的匹配，比较每一个“当前匹配”和它。下面是‘onself’的例子：

```
$longest_match = undef;        # 用于记录最长的匹配

"oneselfsufficient" =~ m{
   one(self)?(selfsufficient)?
   (?{
     # 比较当前匹配($&)与之前记录的最长匹配
     if (not defined($longest_match)
        or
        length($&) > length($longest_match))
     {
        $longest_match = $&;
     }
  })
(?!) # 保证匹配失败，通过回溯继续寻找其他匹配
}x;

# 如果有结果，就输出
if (defined($longest_match)) {
   print "longest match=[$longest_match]\n";
} else {
   print "no match\n";
}
```

毫不奇怪，结果是‘longest match=[oneselfsufficient]’。这一段内嵌代码很长，不过将来我们可能会使用，所以我们把它和「(?!)」封装起来，作为单独的regex对象：

```
my $RecordPossibleMatch = qr{
   (?{
     # 比较当前匹配($&)与之前记录的最长匹配
     if (not defined($longest_match)
         or
         length($&) > length($longest_match))
      {
         $longest_match = $&;
      }
   })
   (?!)  # 保证匹配失败，通过回溯继续寻找其他匹配
}x;
```

下面这个简单例子会找到最长的匹配‘9938’：

```
$longest_match = undef;  # 记录最长的匹配
"800-998-9938" =~ m{ \d+ $RecordPossibleMatch }x;

# 输出到目前为止的累积结果
if (defined($longest_match)) {
    print "longest match=[$longest_match]\n";
} else {
   print "no match\n";
}
```

寻找最左最长的匹配

我们已经能找到最长的全局匹配，现在需要找到出现在最前边的最长匹配。POSIX NFA 就是这样做的（☞177）。所以，如果找到一个匹配，就要禁止传动装置的驱动过程。这样，一旦我们找到某个匹配，正常的回溯会起作用，在同一位置寻找其他可能的匹配（同时需要保存最长的匹配），但是禁用驱动过程保证不会从其他位置寻找匹配。

Perl 不容许我们直接操作传动装置，所以我们不能直接禁用驱动过程，但如果`$longest_match`已经定义，我们能够达到实现禁用驱动过程的效果。测试定义的代码是「`(?{ defined $longest_match})`」，但这还不够，因为它只测试变量是否定义。重要的是根据测试结果进行判断。

在条件判断中使用内嵌代码

为了让正则引擎根据测试结果改变行为，我们把测试代码作为「**(?** *if then* **|** *else***)**」中的 *if* 部分（☞140）。如果我们希望测试结果为真时正则表达式停下来，就把必然失败的「`(?!)`」作为 *then* 部分。（这里不需要 *else* 部分，所以没有出现）。下面是封装了条件判断的 regex 对象：

```
my $BailIfAnyMatch = qr /(?(?{defined $longet_match})(?!))/;
```

if 部分以下画线标注，*then* 部分以粗体标注。下面是它的应用实例，其中结合了前一页定义的`$RecordPossibleMatch`：

```
"800-998-9938" =~ m{ $BailIfAnyMatch \d+ $RecordPossibleMatch }x;
```

得到‘800’，它符合 POSIX 标准——“所有最左位置开始的匹配中最长的匹配”。

在内嵌代码结构中使用 local 函数

Using local in an Embedded-Code Construct

local 在内嵌代码结构中有特殊的意义。理解它需要充分掌握动态作用域（☞295）的概念和第 4 章讲解表达式主导的 NFA 引擎工作原理时所做的“面包渣比喻”（☞158）。下面这段专

门设计（我们会看到，它有缺陷）的程序没有太多复杂的东西，但有助于理解 local 的意义。它检查一行文本是否只包含「\w+」和「\s+」，以及有多少「\w+」是「\d+\b」：

```
my $Count = 0;

$text =~ m{
   ^ (?> \d+  (?{$Count++}) \b | \w+ | \s+ )*$
}x;
```

如果用它来匹配字符串‘123·abc·73·9271·xyz’，$Count 的值是 3。不过，如果匹配字符串‘123·abc·73xyz’，结果就是 2，虽然应该是 1。问题在于，‘73’匹配之后，$Count 的值会发生变化，因为后面的「\b」无法匹配，「\d+」当时匹配的内容需要通过回溯“交还”，内嵌结构的代码却不能恢复到“未执行”的状态。

如果你还不完全了解固化分组「(?>…)」（☞139）和上面发生的回溯也没关系，固化分组用于避免无休止匹配（☞269），但不会影响结构**内部**的回溯，只会影响**重新进入**此结构的回溯。所以如果接下来的「\b」不能匹配，「\d+」的“交还”就完全没有问题。

简单的解决办法是，在$Count 增加之前添加「\b」，保证它的值只有在不进行“交还”操作的情况下才会变化。不过我更愿意在这里使用 local，来说明应用正则表达式期间这个函数对 Perl 代码的影响。来看这段程序：

```
our $Count = 0;

$text =~ m{
 ^ (?> \d+ (?{ local($Count) = $Count + 1 }) \b | \w+ | \s+ )* $
}x;
```

要注意的第一点是，$Count 从 my 变量变为全局变量（我推荐使用 **use strict**，如果这么做了，就必须使用 our 来“声明”全局变量）。

另一点要注意的是，$Count 的修改已经本地化了。关键在于：**对正则表达式内部的本地化变量来说，如果因为回溯需要“交还”local 的代码，它会恢复到之前的值（新设定的值会被放弃）**。所以，即使 local($Count) = $Count +1 在「\d+」匹配‘73’之后执行，把 $Count 的值从 1 改为 2，这个修改也只会是调用 local 时的“本地化到（当前正则表达式的）成功路径”。如果「\b」匹配失败，正则引擎会回溯到 local 之前，$Count 恢复到 1。这也就是正则表达式结束时的值。

内嵌 Perl 代码的插值

因为安全方面的考虑，Perl 不容许用内嵌代码结构「(?{…})」或动态表达式结构「(??{…})」对正则表达式进行字符串插值（不过它们可以进行 regex 对象插值，参考第 334 页的$RecordPossibleMatch），也就是说：

```
m{ (?{ print "starting\n" }) some regex… }x;
```

是可以的，但

```
my $ShowStart = '(?{ print "starting\n" })';
……
m{ $ShowStart some regex… }x;
```

不行。之所以要施加这种限制，是因为把用户的输入作为正则表达式的一部分是长期以来的普遍做法，引入这些结构会容许用户运行任意代码，带来严重的安全隐患。所以，默认情况下不容许这样。

如果你喜欢这样插值，可以使用下面的声明（declaration）：

```
use re 'eval';
```

这就绕开了限制（设置其他参数，编译指示（pragma）use re 也可以用于调试，☞361）

整理用于插值的输入数据

如果采用了上面的做法，而且确实需要使用用户输入的数据插值，请确保其中不包含内嵌 Perl 代码或者动态正则结构。我们可以用正则表达式「\(\s*\?+[p{]」来校验。如果输入数据能够匹配，把它用在正则表达式里就是不安全的。使用「\s*」是因为「/x」修饰符容许开括号之后出现空白字符（我更愿意相信它们不应该出现在那里，不过事实却与此相反）。加号约束的「\?」保证两种结构都可以识别。最后，包含 p 是为了匹配现在已经废弃的「(?p{…})」结构，也就是老版本的「(??{…})」。

我想最好的办法是由 Perl 提供某个修饰符，控制在整个正则表达式或某个子表达式中，容许还是禁止使用内嵌代码。但是在没有实现之前，我们必须按照上面介绍的办法手工检查。

所以，为了保证$Count 的记数不发生错误，必须使用 local。如果把「(?{ print "Final count is $Count.\n" })」放在正则表达式的末尾，它会显示正确的计数值。因为我们希望在匹配完成之后使用$Count，就必须在匹配正式结束之前把它保存到一个非本地化的变量中。因为匹配完成之后，所有在匹配过程中本地化的变量都会丢失。

下面是一个例子：

```
my  $Count = undef;
our $TmpCount = 0;

$text =~ m{
  ^ (?> \d+ (?{ local($TmpCount) = $TmpCount + 1 }) \b | \w+ | \s+ )+ $
  (?{ $Count = $TmpCount })    # 最后将$Count存入非本地变量中
}x;
if (defined $Count) {
   print "Count is $Count.\n";
} else {
   print "no match\n";
}
```

看起来这么做有点儿折腾，但这个例子的目的是说明正则表达式中本地化变量的工作机制。我们会在第 344 页的“模拟命名捕获”中见到实际的应用。

关于内嵌代码和 my 变量的忠告

A Warning About Embedded Code and my Variables

如果 my 变量在正则表达式**之外**声明，那么在正则表达式之中的内嵌代码引用，就必须非常小心， Perl 中变量绑定的详细规定可能会产生重大的影响。在讲解这个问题之前，我必须指出，如果正则表达式的内嵌代码中使用的都是全局变量就没有这种问题，完全可以跳过这一节。忠告：这一节难度不小。

下面的例子说明了问题：

```
sub CheckOptimizer
{
   my $text = shift;      #  第一个参数是要检索的文本
   my $start = undef;     #  记录表达式第一次应用的位置
   my $match = $text =~ m{
     (?{ $start = $-[0] if not defined $start})   #  保存第一次应用的位置
     \d #  这是需要测试的正则表达式
   }x;
   if (not defined $start) {
      print "The whole match was optimized away.\n";
      if ($match) {
          #  这种情况不可能发生!
          print "Whoa, but it matched! How can this happen!?\n";
      }
   } elsif ($start == 0) {
      print "The match start was not optimized.\n";
   } else {
      print "The optimizer started the match at character $start.\n"
   }
}
```

程序中包含 3 个 my 变量，但是只有$start 与此问题有关（因为其他两个并没有在内嵌代

码中引用)。程序首先把$start 设为未定义的值，然后应用开头元素为内嵌代码的匹配，只是在$start 未设定时，内嵌代码结构才会把$start 设置到尝试开始位置。“本次尝试的起始位置”取自$-[0]（@-的第 1 个元素☞302）。

所以，如果调用：

```
CheckOptimizer("test 123");
```

结果就是：

```
The optimizer started the match at character 5.
```

这没有问题，但如果我们再运行一次，结果就成了：

```
The whole match was optimized away.
Whoa, but it matched! How can this happen!?
```

即使正则表达式检查的文本没有变化（而且正则表达式本身也没有变化），结果却不一样了，你发现问题了吗？问题就在于，在第二次调用中编译正则表达式时，内嵌代码中的$start 取的是第一次运行之后设置的值。此函数的其他部分使用的$start 其实是一个新的变量——每次函数调用的开始，执行 my 都会重新设置这个值。

问题的关键就在于，内嵌代码中的 my 变量“锁定”（用术语来说就是：绑定 *bound*）在**具体的** my 变量的实例中，此实例在正则表达式编译时激活。（正则表达式的编译详见 348 页）每次调用 CheckOptimizer，都会创造一个新的$start **实例**，但是用户很难以察觉，内嵌代码中的$start 仍然指向之前的值。这样，函数其他部分使用的$start 实例并没有接收到正则表达式中传递给它的值。

这种类型的实例绑定称为“**闭包**（closure）”，*Programming Perl* 和 *Object Oriented Perl* 之类的书中介绍了这种特性的价值所在。关于闭包，Perl 社群中存在争议，比如本例中闭包究竟是不是一种“特性”，就有不同看法。对大多数人来说，这很难理解。

解决的办法是，不要在正则表达式内部引用 my 变量，除非你知道正则文字的编译与 my 实例的更新是一致的。比如我们知道，第 345 页 SimpleConvert 子程序中使用的 my 变量 $NestedStuffRegex 没有这个问题，因为$NestedStuffRegex 只有一个实例。这里的 my 不在函数或者循环之中，所以它只会在脚本载入时创建一次，然后一直存在，直到程序终止。

使用内嵌代码匹配嵌套结构

Matching Nested Constructs with Embedded Code

328页的程序讲解了如何使用动态表达式匹配任意深度的嵌套结构。一般来说，这都是最简单的方法，但是来看看只使用内嵌代码的办法也没坏处，所以接下来我会给出这种办法。

办法很简单：记录已经遇到的未配对开括号的数量，只有此数量大于0时，才容许出现闭括号。在匹配文本的过程中，我们使用内嵌代码来计数，不过在这之前必须得看看（目前还不能运行的）正则表达式的框架。

```
my $NestedGuts = qr{
  (?>
    (?:
      #  除括号之外的字符
        [^()]+
      #  开括号
      | \(
      #  闭括号
      | \)
    )*
  )
}x;
```

为了保证效率，我们使用了固化分组，因为如果`$NestedGuts`用于更大的正则表达式，就可能导致回溯，这样「`([…]+|…)*`」就会造成无休止匹配（☞226）。举例来说，如果我们将其作为「`m/^\( $NestedGuts \)$/x`」的一部分，应用到‘`(this·is·missing·the·close`’中，如果没有使用固化分组，就得在记录和回溯上花费漫长的时间。

为了配合计数，我们需要4步：

❶ 计数必须从0开始：

```
(?{ local $OpenParens = 0 })
```

❷ 遇到开括号，就把记数器加1，表示有一对括号没有匹配。

```
(?{ $OpenParens++ })
```

❸ 遇到闭括号，就检查记数器，如果大于0，就减去1，表示已经匹配了一对括号。如果等于0，就停止匹配（因为闭括号与开括号不匹配），所以用「`(?!)`」强迫匹配失败。

```
(?(?{ $OpenParens }) (?{ $OpenParens-- }) | (?!) )
```

这里使用了「**(?*if then* | *else*)**」条件判断（☞140），用内嵌代码判断记数器，作为*if*部分。

❹ 一旦匹配结束就检查记数器，确保它等于 0，否则说明仍然有未匹配的开括号，因此匹配失败。

```
(?(?{ $OpenParens != 0 })(?!))
```

综合起来就得到：

```
my $NestedGuts = qr{
   (?{ local  $OpenParens = 0 })   # ❶计算未结束的开括号的数目
   (?> #  固化分组，提高效率
         (?:
          #  除括号以外的字符
            [^()]+
          #  ❷开括号
          | \( (?{ $OpenParens++ })
          #  ❸闭括号
          | \) (?(?{ $OpenParens != 0 }) (?{ $OpenParens-- }) | (?!) )
        )*
   )
   (?(?{ $OpenParens != 0 })(?!)) #  ❹如果还有开括号，则匹配未结束
}x;
```

这段程序的使用方法与第 330 页的`$LevelN`完全相同。

为了分离正则表达式中的`$OpenParens`和程序中可能出现的其他全局变量，这里使用了`local`。但`local`的用法与之前的不同，这里不需要避免回溯，因为正则表达式使用了固化分组，一旦某个多选分支能够匹配，就不会变为“交还”。这样，固化分组既保证了效率，又保证了内嵌代码结构附近匹配的文本不会在回溯中交还（这样`$OpenParens`就与实际匹配的开括号数目一致）。

正则文字重载

Overloading Regex Literals

通过重载，用户可以通过自己喜欢的方式预先处理正则文字中的文字部分。下面几节给出了例子。

添加单词起始/结束元字符

Perl 没有提供作为单词起始/结束元字符的「`\<`」和「`\>`」，可能是因为绝大多数情况下「`\b`」已经够用了。不过，如果我们希望使用这两个元字符，我们可以通过重载，将表达式中的‘`\<`’和‘`\>`’分别替换为「`(?<!\w)(?=\w)`」和「`(?<=\w)(?!\w)`」。

先创建一个函数，MungeRegexLiteral，进行需要的预处理：

```
sub MungeRegexLiteral($)
{
   my ($RegexLiteral) = @_;  #  参数是字符串
   $RegexLiteral =~ s/\\</(?<!\\w)(?=\\w)/g;  #  模拟单词起始边界\<
   $RegexLiteral =~ s/\\>/(?<=\\w)(?!\\w)/g;  #  模拟单词结束边界\>
   return $RegexLiteral;     #  返回修改后的字符串
}
```

如果给此函数传递字符串‘…\<…’，它会将其转化为‘…(?<!\w)(?=\w)…’。记住，因为 replacement 部分类似双引号字符串，所以需要用‘\\w’表示‘\w’。

为了让它能够自动处理正则文字的每个文字部分，我们将其存入文件 *MyRegexStuff.pm*，供 Perl 重载：

```
package MyRegexStuff; #  起个特殊的名字
use strict;       #  这是个好习惯
use warnings;     #  这也是个好习惯
use overload;     #  启用 Perl 的重载机制
 #  载入 regex handler....
sub import { overload::constant qr => \&MungeRegexLiteral }

sub MungeRegexLiteral($)
{
  my ($RegexLiteral) = @_; #  参数是字符串
  $RegexLiteral =~ s/\\</(?<!\\w)(?=\\w)/g;  #  模拟单词起始边界\<
  $RegexLiteral =~ s/\\>/(?<=\\w)(?!\\w)/g;  #  模拟单词结束边界\>
  return $RegexLiteral;  #  返回修改后的字符串
}

1;                         #  标准做法，'use'此文件肯定会返回 true
```

将 *MyRegexStuff.pm* 放在 Perl 的库路径（library path，请参考 Perl 文档中的 PERLLIB）下，所有需要使用此功能的 Perl 脚本都可调用。如果只是为了测试，可以将其放在测试脚本同一目录内，这样调用：

```
use lib '.';               #  在当前目录中寻找库文件
use MyRegexStuff;          #  现在可以使用此功能了
  ......

$text =~ s/\s+\</ /g;      #  将单词之前任意数目任意形式的空白字符替换为单个空格
  ......
```

每个需要这样处理正则文字的程序文件都必须使用 *MyRegexStuff*，但是 *MyRegexStuff.pm* 只需要构建一次（此功能在 *MyRegexStuff.pm* 内部不可用，因为它没有 **use MyRegexStuff**——我们肯定不会这样做）。

添加占有优先量词

我们继续完善 *MyRegexStuff.pm*，让它支持占有优先量词——例如「x++」(☞142)。占有优先量词的作用类似普通的匹配优先量词，只是它们永远不会释放（也就是“交还”）任何已经匹配的内容。用固化分组来模拟的话，只需要去掉最后的‘+’，把量词修饰的所有内容放到固化分组里，「regex*+」就成了「(?>regex*)」(173)。

占有优先量词限定的部分可以是括号内的表达式，也可以是「\w」或者「\x{1234}」之类的元序列，或是普通字符。要处理所有情况并不容易，所以为简便起见，我们只关注作用于括号的?+、*+和++。有了 330 页的$LevelN，我们可以把这段程序：

```
$RegexLiteral =~ s/( \( $LevelN \)[*+?] )\+/(?>$1)/gx;
```

添加到 MungeRegexLiteral 函数。

现在，它成为 overload package 的一部分，我们可以在正则文字中使用占有优先量词，例如第 198 页的这个例子：

```
$text =~ s/"(\\.|[^"])*+"//;        # 去掉双引号字符串
```

如果要处理的情况不只是括号，就要复杂很多，因为正则表达式中的变数很多，下面是一种尝试：

```
$RegexLiteral =~ s{
    (
      #  匹配可能的限定对象
      (?: \\[\\abCdDefnrsStwWX]   #  \n、\w 之类
        | \\c. #  \cA
        | \\x[\da-fA-F]{1,2}      #  \xFF
        | \\x\{[\da-fA-F]*\}      #  \x{1234}
        | \\[pP]\{[^{}]+\}        #  \p{Letter}
        | \[\]?[^]]+\]            #  字符组
        | \\\W                    #  \*
        | \( $LevelN \)           #  (…)
        | [^()*+?\\]              #  其他任何字符
       )
      #  ...标准量词...
      (?: [*+?] | \{\d+(?:,\d*)?\} )
   )
   \+ #  ..和量词之后的‘+’
}{(?>$1)}gx;
```

这个表达式的大体形式和之前一样：使用占有优先量词匹配一些内容，去掉最后的‘+’，将整个表达式用「(?>…)」围起来。要想识别 Perl 正则表达式的复杂语法，这样还很不够。匹配字符组的部分亟需改进，因为它并不能识别字符组内部的转义。更糟糕的是，这个表达式的基本思路有问题，因为它不能完整识别 Perl 的正则表达式。比如，它就不能正确处理‘\(blah\)++’中作为普通字符的开括号，而是认为「++」仅仅限定「\)」。

解决这个问题得花许多工夫，或许得想办法从前往后仔细遍历整个正则表达式（类似第132页的补充内容中的办法）。我本来希望改善处理字符组的元素，但是最后觉得没必要处理其他复杂情况，原因有两个。第一个是，这个表达式能应付大部分正常的情况，所以修正处理字符组的元素就能满足实用要求了。更重要的一点是，目前 Perl 的正则表达式重载有严重问题，结果它的用途大打折扣，讨论见下一节。

正则文字重载的问题

Problems with Regex-Literal Overloading

正则文字重载的功能非常有用，至少在理论上是如此，不幸的是实际情况并非如此。问题在于，它只对正则文字中的文字部分有效，而不会影响插值部分。例如，在 **`m/($MyStuff)*+/`** 中 `MungeRegexLiteral` 函数调用了两次，一次是在变量插值之前（“**(**”）；另一次是插值之后（“**)*+**”）。（它永远不会影响`$MyStuff` 的值）。因为重载必须同时找到两个部分，而插入的值又是不确定的，所以实际上重载不会生效。

对之前添加的`\<`和`\>`来说，这不是个问题，因为变量替换不太可能把它们切段。但是因为重载不会影响插值变量，包含‘\<’或‘\>’的字符串或 regex 对象就不会受重载影响。上一节已经提到，如果由重载来处理正则文字，就很难每次都保证完整性和准确性。即使是与`\>`一样简单也会出问题，例如‘`\\>`’，它表示反斜线‘`\`’之后紧跟尖括号‘`>`’。

另一个问题是，重载不知道正则表达式所使用的修饰符。表达式是否使用了`/x` 是很重要的问题，但重载没有确切的办法知道。

最后还必须指出，使用重载会禁止根据 Unicode 命名指定字符的功能（「`\N{`*name*`}`」☛290）。

模拟命名捕获

Mimicking Named Capture

讲完了重载的不便之后，我们来看看综合了许多特殊结构的复杂例子。Perl 没有提供命名捕获（☛138）的功能，但是我们可以使用捕获型括号和`$^N` 变量（☛301）来模拟，这个变量引用的是最近结束的捕获型括号匹配的内容（现在我假扮 Perl 开发人员，使用`$^N`，特意为 Perl 增加命名捕获的功能）。

来看个简单的例子：

```
「href\s*=\s*($HttpUrl)(?{ $url = $^N })」
```

这里使用了 303 页的 regex 对象$HttpUrl。下画线部分是一段内嵌代码，把$HttpUrl 匹配的内容保存到$url 中。在这里用$^N 取代$1 似乎有些多此一举，甚至不必要使用内嵌代码，因为在匹配之后使用$1 更加方便。但是如果把其中一部分封装到 regex 对象，然后多次使用：

```
my $SaveUrl = qr{
   ($HttpUrl)              # 匹配 HTTP URL...
   (?{ $url = $^N })       # ...保存到$url
}x;

$text =~ m{
   http \s*=\s* ($SaveUrl)
  | src \s*=\s* ($SaveUrl)
}xi;
```

无论$HttpUrl 是怎么匹配的，$url 都会被设置为 URL。在这个简单应用中可以使用其他办法（例如$+变量☞301），但是在更复杂的情况中，$SaveUrl 之外的办法更难维护，所以将它保存到命名变量中方便得多。

这里有一个问题，如果设定$url 的结构在回溯中被"交还"，已设定的值却不会"撤销保存（unwritten）"。所以要在初始匹配时修改本地化的临时变量，只有在整体匹配真正确认之后才保存"真正"的变量，就像第 338 页的例子一样。

下面给出了一种解决办法。从用户的角度来看，在「(?<Num>**\d+**)」之后，「\d+」匹配的数值仍然可以以$^N{**Num**}访问。尽管未来版本的 Perl 可能会把%^N 转换为某种特殊的系统变量，现在仍然不是特殊的，所以我们可以随意使用。

我们可以使用%NamedCapture 之类的名字，但选择%^N 是有理由的。之一是它类似$^N。另一个理由是，如果写明了 use strict，它不需要预声明。最后，我希望 Perl 最终会内建对命名捕获的支持，所以我认为%^N 是个好办法。如果果真如此，%^N 就能够和正则表达式的其他变量（☞299）一样，自动使用动态作用域。但是目前，它只是普通的全局变量，所以不会自动使用动态作用域。

当然，即便是这个程序也会出现正则文字重载的办法所具有的问题，例如不能处理插值变量。

模拟命名捕获

```
package MyRegexStuff;
use strict;
use warnings;
use overload;
sub import { overload::constant('qr' => \&MungeRegexLiteral) }

my $NestedStuffRegex;  #  在自身定义中使用，必须预声明
$NestedStuffRegex = qr{
(?>
    (?:      #  非括号非转义的字符...
           [^()\#\\]+
           #  转义字符...
       | (?s: \\. )
           #  正则表达式注释...
       | \  #.*\n
           #  匹配嵌套结构...
       | \( (??{ $NestedStuffRegex }) \)
  )*
)
}x;
sub SimpleConvert($); #  递归调用，必须预声明
sub SimpleConvert($)
{
  my $re = shift; #  要处理的表达式
  $re =~ s{
      \(\? # "(?"
        < ( (?>\w+) ) >           #  <$1 > $1 是标识符
      ( $NestedStuffRegex )       #  $2 - 可能出现的嵌套结构
      \)                          #  ")"
  }{
    my $id = $1;
    my $guts = SimpleConvert($2);
    # 把
    # (?<id>guts)
    # 改为
    # (?: (guts) # 配 guts
    # (?{
    #     local($^N{$id}) = $guts  #  保存 %^T 中的本地化元素
    # })
    # )
    "(?:($guts)(?{ local(\$^T{'$id'}) = \$^N }))"
  }xeog;
  return $re;                        #  返回处理结果
}

sub MungeRegexLiteral($)
{
  my ($RegexLiteral) = @_;    #  参数为字符串
  # print "BEFORE: $RegexLiteral\n";           #  调试时取消注释
  my $new = SimpleConvert($RegexLiteral);
  if ($new ne $RegexLiteral)
```

```
{
   my $before = q/(?{ local(%^T) = () })/; # 本地化临时 hash 变量
   my $after = q/(?{ %^N = %^T })/;        # 把它们拷贝到"真正的"hash 变量
   $RegexLiteral = "$before(?:$new)$after";
}
# print "AFTER: $RegexLiteral\n";          # 调试时取消注释
return $RegexLiteral;
}
1;
```

效率

Perl Efficiency Issues

在大多数情况下，Perl 中正则表达式的效率问题与任何使用传统 NFA 的工具一样。第 6 章介绍的技巧——内部优化、消除循环，以及“开动你的大脑”，都适用于 Perl。

当然， Perl 也有专属于自己的问题，这一节我们就来看看：

- **办法不止一种** Perl 就像一个工具箱，同一种问题可以用许多办法来解决。理解了 Perl 的思维方式，就会明白哪些问题是钉子，但是选择合适的锤子还需要花很多工夫来编写高效而易于理解的程序。有时候，效率和易于理解似乎是不相容的，不过一旦理解深入了，就能做出更好的选择。

- **表达式编译、qr/…/、/o 修饰符和效率** 正则运算符的编译和插值，做得好的话能节省大量的时间。/o 修饰符还没有详细讲解，它配合 regex 对象(qr/…/)，能够调控耗费时间的重编译过程。

- **$&的负面影响** 伴随效应设定的变量$`、$&和$'，也许很方便，但存在不易发现的效率陷阱，哪怕只出现了一次，也可能带来麻烦。所以并不是非得使用它们——只要脚本中出现了任意一个变量，负面影响就不可避免。

- **Study 函数** 近年来，Perl 提供了 study(…)函数。按照预期，它能提高正则表达式的速度，但是似乎没有人真正知道它是否能提高速度，以及背后的原因。

- **性能测试** 性能测试的规矩就是，越快的程序终止得越早（你可以引用我的话）。无论小型函数、大型函数，还是处理真实数据的整个程序，性能测试都是判断速度的最终标准。尽管性能测试有各种各样的办法，Perl 中的性能测试却是简单而轻松的。我会给出我用的办法，这个办法在写作本书时做过数百次性能测试。

- **正则表达式调试** Perl 的正则表达式调试标识位（debug flag）可以告诉我们，正则引擎和传动装置对正则表达式进行了哪些优化。下面会讲解如何查看这些信息，以及 Perl 包含了哪些秘密。

办法不只一种

"There's More Than One Way to Do It"

通常，一个问题总是有许多种解法，所以在权衡效率和可读性时，应该做的就是了解所有的办法。来看个简单的问题，修改一个 IP 地址，例如‘18.181.0.24’，保证每一段都包含三位数字：‘018.181.000.024’。简单的办法是：

```
$ip = sprintf("%03d.%03d.%03d.%03d", split(/\./, $ip));
```

这办法当然没错，但显然还有其他的解决办法。表 7-6 列出了好几种办法，比较了它们的效率（按照效率排序，最上面的效率最高）。这个例子的目的很简单，本身也没有太多价值，但是它能代表简单的文本处理任务，所以我鼓励你花一点时间理解各种办法。可能有些技巧你没见过。

如果输入格式正确的 IP，每个办法都能到正确的结果，但是如果输入别的数据则可能会出错。如果数据是不规范的，可能就需要多花点心思。除此之外，实际差别在于效率和可读性。就可读性而言，#1 和#13 似乎是最好理解的（尽管效率上存在巨大的差异）。同样易于理解的是#3 和#4（类似#1），以及#8（类似#13）。其他解法都太过复杂了。

那么效率呢？为什么不同的解法有不同的效率？原因在于 NFA 的工作原理（第 4 章），Perl 的各种正则优化措施（第 6 章），以及其他 Perl 结构的处理速度（例如 sprintf，以及 substitution 运算符的机制）。substitution 运算符的/e 修饰符，有时候虽然不可或缺，但效率低的解法似乎都使用了它。

比较#3 和#4，#8 和#14 很有意义。这两对正则表达式的区别只是在于括号——没有括号的表达式要比有括号的稍快一点。#8 使用$&来避免括号带来的高昂代价，性能测试却无法体现这一点（☞355）。

表达式编译、/o 修饰符、qr/…/和效率

Regex Compilation, the /o Modifier, qr/ /, and Efficiency

Perl 中与表达式效率相关的另一个重点是，程序遇到正则运算符之后，在实际应用正则表达

表 7-6：填补 IP 地址的若干解法

<table>
<tr><th>排名</th><th>耗时</th><th>解法</th></tr>
<tr><td>1</td><td>1.0X</td><td><code>$ip = sprintf("%03d.%03d.%03d.%03d", split(m/\./, $ip));</code></td></tr>
<tr><td>2</td><td>1.3X</td><td><code>substr($ip, 0, 0) = '0' if substr($ip, 1, 1) eq '.';</code>
<code>substr($ip, 0, 0) = '0' if substr($ip, 2, 1) eq '.';</code>
<code>substr($ip, 4, 0) = '0' if substr($ip, 5, 1) eq '.';</code>
<code>substr($ip, 4, 0) = '0' if substr($ip, 6, 1) eq '.';</code>
<code>substr($ip, 8, 0) = '0' if substr($ip, 9, 1) eq '.';</code>
<code>substr($ip, 8, 0) = '0' if substr($ip,10, 1) eq '.';</code>
<code>substr($ip, 12, 0) = '0' while length($ip) < 15;</code></td></tr>
<tr><td>3</td><td>1.6X</td><td><code>$ip = sprintf("%03d.%03d.%03d.%03d", $ip =~ m/\d+/g);</code></td></tr>
<tr><td>4</td><td>1.8X</td><td><code>$ip = sprintf("%03d.%03d.%03d.%03d", $ip =~ m/(\d+)/g);</code></td></tr>
<tr><td>5</td><td>1.8X</td><td><code>$ip = sprintf("%03d.%03d.%03d.%03d",</code>
<code>$ip =~ m/^(\d+)\.(\d+)\.(\d+)\.(\d+)$/);</code></td></tr>
<tr><td>6</td><td>2.3X</td><td><code>$ip =~ s/\b(?=\d\b)/00/g;</code>
<code>$ip =~ s/\b(?=\d\d\b)/0/g;</code></td></tr>
<tr><td>7</td><td>3.0X</td><td><code>$ip =~ s/\b(\d(\d?)\b)/$2 eq '' ? "00$1" : "0$1"/eg;</code></td></tr>
<tr><td>8</td><td>3.3X</td><td><code>$ip =~ s/\d+/sprintf("%03d", $&)/eg;</code></td></tr>
<tr><td>9</td><td>3.4X</td><td><code>$ip =~ s/(?:(?<=\.)|^)(?=\d\b)/00/g;</code>
<code>$ip =~ s/(?:(?<=\.)|^)(?=\d\d\b)/0/g;</code></td></tr>
<tr><td>10</td><td>3.4X</td><td><code>$ip =~ s/\b(\d\d?\b)/'0' x (3-length($1)) . $1/eg;</code></td></tr>
<tr><td>11</td><td>3.4X</td><td><code>$ip =~ s/\b(\d\b)/00$1/g;</code>
<code>$ip =~ s/\b(\d\d\b)/0$1/g;</code></td></tr>
<tr><td>12</td><td>3.4X</td><td><code>$ip =~ s/\b(\d\d?\b)/sprintf("%03d", $1)/eg;</code></td></tr>
<tr><td>13</td><td>3.5X</td><td><code>$ip =~ s/\b(\d{1,2}\b)/sprintf("%03d", $1)/eg;</code></td></tr>
<tr><td>14</td><td>3.5X</td><td><code>$ip =~ s/(\d+)/sprintf("%03d", $1)/eg;</code></td></tr>
<tr><td>15</td><td>3.6X</td><td><code>$ip =~ s/\b(\d\d?(?!\d))/sprintf("%03d", $1)/eg;</code></td></tr>
<tr><td>16</td><td>4.0X</td><td><code>$ip =~ s/(?:(?<=\.)|^)(\d\d?(?!\d))/sprintf("%03d", $1)/eg;</code></td></tr>
</table>

式前，Perl 必须在幕后进行预处理工作。真正的准备工作依赖于正则运算元的类型。在大多数情况下，正则运算元是正则文字，例如 `m/…/`、`s/…/…/` 或 `qr/…/`。Perl 必须对它们进行幕后处理，而处理需要的时间，如果可能应该尽力避免。首先，让我们来看要做的事情，然后讲解如何避免。

预处理正则表达式的内部机制

预处理正则运算元的机制在第 6 章有所涉及（241），不过 Perl 还有自己的处理。

Perl 对正则运算符的预处理大致分为两步：

1. **正则文字处理** 如果运算符是正则文字，就按照“正则文字的解析方式”（☞292）中的描述来处理。变量插值就发生在这一步。

2. **正则编译** 检查正则表达式，如果符合规则，就将其编译为适用于正则引擎实际应用的内在状态（如果不符合规则，则报错）。

正则表达式编译完成之后，就能够实际应用到目标字符串中，参见第 4 到第 6 章。

并不是每使用一次正则运算符，就需要进行一次预处理。但是正则文字**第一次**使用时，必须进行预处理，但如果多次执行同样的正则文字（例如在循环中，或是调用多次的函数），Perl 有时候能够重用之前的工作。下一节说明了 Perl 如何做到这一点，以及程序员可以使用的提高效率的技巧。

减少正则编译的步骤

下面几节中我们会见到 Perl 避免某些正则文字相关预处理的两种办法：**无条件缓存**（unconditional caching）和**按需重编译**（on-demand recompilation）。

无条件缓存

如果正则文字中没有插值变量，Perl 就知道这个正则表达式在两次应用之间不会变化，所以第一次编译完成之后，会保存编译的形式（“缓存”），以备将来使用。无论正则表达式会执行多少次，只需要检查和编译一次。本书中的大多数正则表达式都没有变量插值，因此从这个方面来说效率无可挑剔。

内嵌代码和动态正则结构中的变量则不属于此类，因为它们不会**插值**到正则表达式中，而是作为正则表达式执行的固定代码的一部分。有时候，你可能希望每次执行都解释内嵌代码中引用的 my 变量，请不要忘记第 338 页的忠告。

有一点要说清楚，缓存的持续时间与代码的执行时间相同，下次运行同样代码时不会有任何的缓存。

按需重编译

并不是所有的正则运算元都能够直接缓存，比如下面的代码：

```
my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];
#  $today 保存的是星期数("Mon", "Tue"之类)

while (<LOGFILE>) {
    if (m/^$today:/i) {
```

`m/^$today:/`中的正则表达式需要插值，虽然在循环中使用，但每轮循环的插值结构是相同的。所以一再重复编译同样的表达式的效率很低，所以 Perl 会自动进行简单的字符串检查，比较本次和上次插值的结果。如果相同，就使用上次的缓存。如果不同，就重新编译正则表达式。所以，对比缓存值并重新插值尽可能避免了相对更耗时的编译。

这样究竟能节省多少时间呢？非常多。举例来说，我测试了第 303 页的`$HttpUrl`（使用扩展的`$HostnameRegex`）的三种预处理方式。设计的性能测试能准确体现预处理的开销（使用插值、字符串检查、编译，以及其他后台任务），而不是表达式应用的整体开销，因为在任何情况下这种应用的时间都是一样的。

结果非常有意思。我运行了没有插值的版本（整个正则表达式都硬编码在 m/…/中），用它作为比较的基础。如果正则表达式每轮循环不会改变，比较并插值大概需要 25 倍的时间。完整的预处理（每轮循环都要重新编译）大概需要 1 000 倍的时间，这数字真惊人！

应用到实际场合就会发现，完整的预处理即使比静态正则文字预处理要慢 1 000 倍，在我的机器上也只需要大约 0.000 26 秒（测试的速度是每秒 3 846 次，相反，静态正则文字预处理的速度是每秒 370 万次）。当然，不使用插值能够节省的时间非常可观，不进行重编译节省的时间显然也很可观。下面几节，我们会考察如何在更多情况下使用这些技巧。



表示“一次性编译”的/o修饰符

简单地说，如果正则文字运算元中使用了/o修饰符，它就会只会检查和编译一次，而无论是否包含插值。如果没有插值，添加/o不会有任何改变，因为没有插值的表达式总是会自动缓存。但如果使用了插值，程序执行第一次遇到正则文字时，会进行正常的完整的预处理，但因为/o的存在，内在状态会存储下来。如果之后又遇到这个正则运算元，就会直接调用缓存。

下面这个例子之前也出现过，只是现在添加了/o：

```
my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

while (<LOGFILE>) {
    if (m/^$today:/io) {
```

这个表达式要快得多，因为从第二次开始的每轮循环中，正则表达式都忽略了$today。不需要插值，也不需要预处理和重新编译正则表达式，能够节省大量的时间，而这是Perl无法自动完成的，因为使用了变量插值，$today可能会变化，所以为了安全，Perl必须每次都检查。使用/o就告诉Perl，在第一次预处理和编译完成之后“锁定”这个表达式。因为我们知道，插值变量是不变的，即使变化了，也不希望使用新值，所以这样做完全没问题。

/o的潜在“陷阱”

在使用/o时，有个重要的“陷阱”必须要注意。例如下面这个函数：

```
sub CheckLogfileForToday()
{
  my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

  while (<LOGFILE>) {
      if (m/^$today:/io) {     # 危险——这里要小心
          ......
      }
  }
}
```

记住，/o表示正则运算元只需要编译一次。第一次调用CheckLogfileForToday()时，代表当天日期的正则运算元就锁定在其中。如果过了一段时间再次调用这个函数，即使$today变化了，也不会重新检查；在程序执行过程中，每次使用的都是最开始锁定在其中的正则表达式。

这个问题很严重，不过下一节中，regex 对象提供了两全其美的解决办法。

用 regex 对象提高效率

迄今为止，我们看到的所有关于预处理的讨论都适用于正则文字。其目的在于花尽可能少的工夫获得编译好的正则表达式。达到此目的的另一个办法是使用 regex **对象**，把编译好的正则表达式封装在变量内部供程序使用。可以使用 qr/…/创建 regex 对象（☞303）。

下面是使用 regex 对象的例子：

```
sub CheckLogfileForToday()
{
  my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

  my $RegexObj = qr/^$today:/i;   #  每次调用编译一次

  while (<LOGFILE>) {
      if ($_ =~ $RegexObj) {
            ......
      }
  }
}
```

每调用一次函数，就会创建一个新的 regex 对象，但是之后它只是直接用于 log 文件的每一行。如果 regex 对象用做运算元，它不会进行前面介绍的任何预处理。预处理是在 regex 对象**创建**而不是使用时进行的。可以把 regex 对象想象为“自动设定的正则缓存”，这个编译好的表达式可以在任何地方使用。

这个办法兼具了两方面的优点：它效率高，因为只有在每次函数调用（而不是 log 文件的每一行）时才会编译，但是，与之前错误使用/o 的例子不同，即使多次调用 CheckLogfile-ForToday()，也没有问题。

需要弄清楚的是，这个例子中出现了两个正则运算元。正则运算元 qr/…/并不是一个 regex 对象，但能从接收的正则文字**创建** regex 对象。然后这个对象用作循环中 match 运算符=~的运算元。

regex 对象配合 m/…/

这段程序：

```
if ($_ =~ $RegexObj) {
```

也可以这样写：

```
if (m/$RegexObj/) {
```

此时已经不是普通的正则文字了，尽管看上去没有区别。“正则文字”的内容就是 regex 对象，它与直接使用 regex 对象一样。这种做法的好处在于：m/…/更为常见，更容易使用。也不用明确指定目标字符串$_，方便与其他使用同样默认变量的运算符结合。最后一个原因是，这样我们能够对 regex 对象使用/g。

/o 配合 qr/…/

/o 修饰符可以配合 qr/…/，但在这里你肯定不希望如此。就像用/o 配合其他任何正则运算符一样，qr/…/o 在第一次使用正则表达式时就会进行锁定，所以如果这样写，无论$today 如何变化，每次调用这个函数$RegexObj 使用的都是同样的 regex 对象。这与第 352 页的 m/…/o 的问题一样。

依靠默认表达式提高效率

正则运算符的默认表达式（☞308）可以提高效率，尽管使用 regex 对象可能更划算。不过我还是会简要介绍一番。例如：

```
sub CheckLogfileForToday()
{
  my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

  # 直到匹配为止，设置默认表达式
  "Sun:" =~ m/^$today:/i or
  "Mon:" =~ m/^$today:/i or
  "Tue:" =~ m/^$today:/i or
  "Wed:" =~ m/^$today:/i or
  "Thu:" =~ m/^$today:/i or
  "Fri:" =~ m/^$today:/i or
  "Sat:" =~ m/^$today:/i;

   while (<LOGFILE>) {
      if (m//) {  #  使用默认的表达式
         ......
      }
   }
}
```

使用默认正则表达式的关键在于，只有匹配成功才会设置默认值，所以$today 设置之后还有长长的代码。你已经看到，这相当不美观，所以我不推荐这么做。

理解“原文”副本

Understanding the "Pre-Match" Copy

在匹配和替换时，Perl 有时必须动用额外的空间和时间来保存目标字符串在匹配之前的副本。我们会看到，有时这个副本会用于支持重要特性，有时则不会。应该尽量避免不会用到的副本，提高效率，尤其是在目标字符串很长，或者速度非常重要的情况下更是如此。

下一节我们会讲解何时以及为什么 Perl 可能会保存目标字符串的原文副本，什么时候用到副本，以及在效率极端重要时，如何取消这个副本来提高效率。

通过原文副本支持$1、$&、$'、$+…

对于 match 或者 substitution 操作的目标字符串，Perl 会生成一个原文副本，以支持$1、$&之类匹配后的变量（☞299）。每次匹配完成之后，Perl 不会实际生成这些变量，因为许多变量（还有可能是所有）根本不会被程序用到。相反，Perl 只是保存原始字符串的副本，记住各种匹配发生在原来文本中的位置，在使用$1 之类变量时通过位置来引用。这种办法不错，工作量小，因为多数情况下都不会用到某些甚至全部的匹配后的变量。这种“延迟求值（lazy evaluation）”能避免许多不必要的工作。

尽管延迟创建$1 之类的变量能够节省工作量，但保存目标字符串的副本仍然需要成本。而这是必要的吗？为什么不能直接使用原来的文本？请参考：

```
$Subject =~ s/^(?:Re:\s*)+//;
```

这样，$&正确地引用了$Subject 中删除的文本，但因为它已经从$Subject 中删除，在后面用到$&时，Perl 不能从$Subject 中引用这段文本。下面的代码情况相同：

```
if ($Subject =~ m/^SPAM:(.+)/i) {
    $Subject = "-- spam subject removed --";
    $SpamCount{$1}++;
}
```

引用$1 时，原来的$Subject 已经删除了。所以，Perl 必须保存原文副本。

原文副本并非时时必须

在实践中，原文副本的主要“用户”是$1、$2、$3 之类的变量。但是如果正则表达式不包

含捕获型括号呢？那样就不必担心$1 之类了，所以完全不必考虑如何支持它们。所以至少，不包含捕获型括号的正则表达式可以不必保存拷贝？答案是未必。

不宜使用的变量：$`、$&和$'

$`、$&、$'这三个变量与捕获型括号无关。它们分别对应到匹配文本之前的部分，匹配文本和匹配之后的部分，其实可以应用于**每一次** match 和 substitution。Perl 不能预先知道某个匹配中是否会用到这些变量，所以**每次都必须保存原文副本**。

听起来，似乎没有办法省略副本，但是 Perl 足够聪明，它能够认识到，如果这些变量不会出现在程序中，就根本没必要（甚至在任何可能用到的 library 之中）保存副本来提供支持。所以，**如果没有用到捕获型括号，再避免出现$`、$&和$'就能省略原文副本——这是很棒的优化！**只要在程序中的任何一处用到了$`、$&和$'三者之一，整个优化即告失效。这可不够意思！所以，我认为这些变量是“不宜使用（naughty）”的。

原文副本的代价有多高

我进行了简单的性能测试，对 Perl 源代码的 130 000 行 C 程序中的每一行检查 m/c/。这个性能测试仅仅检测哪一行出现了字符‘c’。测试的目的是衡量原文副本的影响。我用两种方法进行测试：一种肯定没有用到原文副本，一种肯定用到了。因此，唯一的区别就在于保存副本的开销。

保存原文副本的程序所用的时间要长 40%。这代表了“平均最差情况”，这样说是因为性能测试并没有进行什么实质性的操作，否则二者之间的时间相对比例会减小（甚至显得微不足道）。

另一方面，在真正的最坏情况下，额外副本可能真的占据非常重要的比重。我对同样的数据运行同样的程序，但是这次将所有超过 3.5MB 的数据都放在一行中，而不是长度合适的 130 000 行。这样就能比较单次匹配的相对表现。不使用原文副本的匹配几乎是立刻就得到了结果，因为第一个‘c’字符离开头不远，匹配之后程序就运行结束。而使用原文副本的程序运行原理差不多，只是它会首先拷贝整个字符串。它所用的时间大约是前者的 7 000 倍。因此我们知道，避免使用某些结构能够提高效率。

避免使用原文副本

如果 Perl 能够领会程序员的意图，只在需要的情况下保存副本，当然很好。但请记住，这些副本并不是“败笔”——Perl 在幕后处理这些繁琐事务是我们选择 Perl，而不是 C 或者汇编语言的原因。事实上，Perl 最初只是为了把用户从繁杂的机制中解放出来，这样他们只需要关注问题的解决方案就好了。

永远不要使用不宜使用的变量。同样，尽可能避免额外的工作也是不错的。首先想到的就是，永远不要在代码中使用`$``、`$&`和`$'`。通常，`$&`很容易消除——把正则表达式包围在一个捕获型括号内，然后使用`$1` 即可。举例来说，把 HTML tag 转换为小写时，不使用 **`s/<\w+>/\L$&\E/g`**，而使用 **`s/(<\w+>)/\L$1\E/g`**。

如果保存了原始目标字符串，就可以很容易地模拟`$``和`$'`。匹配某个 *target* 字符串之后，可以按下面的规则来模拟：

变量	模拟方式
`$``	substr(***target***, 0, $-[0])
`$&`	substr(***target***, $-[0], $+[0]-$-[0])
`$'`	substr(***target***, $+[0])

因为`@-`和`@+`（302）保存的是原始目标字符串中的**位置**，而不是确切的**文本**，使用它们不需要担心效率问题。

我还给出了`$&`的模拟。相对使用捕获型括号和`$1` 的办法，这可能是一个更好的办法，这样完全不必使用任何捕获型括号。请记住，避免使用`$&`之类变量的目的就在于，如果表达式中没有出现捕获型括号，要避免保存原始副本。如果修改程序，去掉`$&`，再对每个匹配都增加捕获型括号，就不会节省任何时间。

不要使用不宜使用的模块。当然，避免`$``、`$&`、`$'`也意味着避免使用调用它们的模块。Perl 的核心模块中，除 `English` 之外都没有使用它们。如果你希望使用 `English` 模块，可以这样：

```
use English '-no_match_vars';
```

这样就没有问题了。如果你从 CPAN 或者其他地方下载了模块，你可能需要检查一番，看看它们是否使用了这些变量。请参考下一页的补充内容，里面有些诀窍，告诉你如何判断程序是否用到了这些变量。

如何检查代码是否包含$&

判断程序是否使用了不宜使用的变量$`、$&和$'，并不是件容易的事情，尤其使用库函数时更是如此，但还是有几个办法来判断。最简单的是，如果你的二进制程序在编译时指定了-DDEBUGGING参数，就可以使用**-c**和**-Mre=debug**参数（☞361），观察输出的结尾，找到包含‘**Enabling $`、$&和$' support**’或者‘**Omitting $`、$&和$'、support**’的那一行。如果见到前面的文字，就表示使用了这些变量。

另一种可能（但不太现实）的情况是，程序在eval语句中使用了这些变量，如果没有实际运行，Perl不知道的是否使用了这些变量。解决办法之一就是安装CPAN（*http://www.cpan.org*）的Devel:SawAmpersand Package：

```
END {
   require Devel::SawAmpersand;
   if (Devel::SawAmpersand::sawampersand) {
       print "Naughty variable was used!\n";
   }
}
```

与Devel::SawAmpersand一起的还有Devel::FindAmpersand，这个package能够告诉用户有问题的代码的位置。不幸的是，对最新版本的Perl，它不能保证完全正确。这两个package的安装都不简单，所以要做的事没准很多（请参考*http://regex.info/*查找可能的更新）。

用查找性能损失的办法找出问题代码的办法也值得一看：

```
use Time::HiRes;
sub CheckNaughtiness()
{
  my $text = 'x' x 10_000;  #  创建一定量的数据

  #  计算纯循环的开销
  my $start = Time::HiRes::time();
  for (my $i = 0; $i < 5_000; $i++) { }
  my $overhead = Time::HiRes::time() - $start;

  #  计算同样次数匹配的开销
  $start = Time::HiRes::time();
  for (my $i = 0; $i < 5_000; $i++) { $text =~ m/^/ }
  my $delta = Time::HiRes::time() - $start;

  #  计算差值
  printf "It seems your code is %s (overhead=%.2f, delta=%.2f)\n",
     ($delta > $overhead*5) ? "naughty" : "clean", $overhead, $delta;
}
```

Study 函数

The Study Function

与优化正则表达式本身不同，study(…)优化了对特定字符串的某些检索。一个字符串在study之后，应用到它的（一个或多个）正则表达式可以从缓存的分析数据中受益。一般是这样使用的：

```
while (<>)
{
   study($R);      # 匹配之前Study默认的目标字符串$_
   if (m/regex 1/) { … }
   if (m/regex 2/) { … }
   if (m/regex 3/) { … }
   if (m/regex 4/) { … }
}
```

study的作用很简单，但是理解它什么情况下有价值却不简单。它不会影响到程序的任何值和任何结果，唯一的影响就是，Perl会使用更多的内存，总的执行时间可能会增加，保持不变，或者减少（这是我们预期的）。

study一个字符串时，Perl会分配时间和内存来记录字符串中的一系列位置（在大多数系统中，需要的内存是字符串大小的4倍）。在字符串修改之前，针对此字符串的每次匹配都可以从中受益。对字符串的任何修改都会导致study数据的失效，相当于study另一个字符串。

Study能给目标字符串提供的帮助，很大程度上取决于用来匹配的正则表达式，以及Perl能够使用的优化。例如用m/foo/检索文本，如果使用了study，速度会提升很多（如果字符串更长，甚至可能提高10000倍）。但是，如果使用了/i，就不会有这种效果，因为/i不会利用study的结果（和其他优化）。

不应该使用study的情况

- 如果只使用/i，或是所有正则文字都受「(?i)」或「(?i:…)」作用，就不应该对字符串使用study，因为它们不能从study中受益。
- 如果目标字符串很短，也不应该使用study。因为此时，正常的固定字符串识别优化（☞247）已经足够了。那么“短”究竟如何界定呢？字符串的长度没有确切的标准，所以具体来说，只有进行性能测试才能判断study是否有益。不过权衡利弊，我通常不使用study，除非字符串的长度为若干KB。

如果你只希望在修改之前，或是 study 不同的字符串之前，对目标字符串进行少数几次匹配，请不要使用 study。如果要获得真正的性能提升，必须是多次匹配节省下来的时间长于 study 的时间。如果匹配次数较少，花在 study 身上的时间抵不上节省的时间，得不偿失。

只对期望使用包含“独立出来的”文字（☞255）的正则表达式搜索的字符串使用 study。如果不知道匹配中必须出现的字符，study 就派不上用场（看了这几条，也许你会认为，study 对 index 函数有益，但事实并非如此）。

什么时候使用 study

最适合使用 study 的情况就是，目标字符串很长，在修改之前会进行许多次匹配。一个简单的例子就是我在写作本书时所用的过滤器。我用自己的标记法写稿，然后用过滤器转换为 SGML（再转换为 *troff*，再转换为 PostScript）。经过过滤器内部，一整章变为一个大字符串（例如，本章的大小为 475KB）。在退出之前进行多项检查来保证不会漏过错误的标记。这些检查不会修改字符串，它们通常查找固定长度的字符串，所以它们很适合于 study。

性能测试

Benchmarking

如果你真的关心效率，最好的办法就是进行性能测试。Perl 自带的 Benchmark 模块提供了详细的文档（"perldoc Benchmark"）。可能是习惯使然，我更喜欢从自己动手写性能测试：

```
use Time::HiRes 'time';
```

我把希望测试的内容简单包装成：

```
my $start = time;
my $delta = time - $start;
printf "took %.1f seconds\n", $delta;
```

性能测试的重要问题包括，确保性能测试进行了足够多的工作，显示的时间真正有意义，尽可能多地测试希望的部分，尽可能少地测试不希望的部分。在第 6 章有详细的讲解（☞232）。找到正确的测试方法可能得花些时间，但是结果可能非常有价值，也很值得。

正则表达式调试信息

Regex Debugging Information

Perl 提供了数量众多的优化措施，期望能够尽可能快地找到匹配；第 6 章的“常见优化措施”（☞204）介绍了基础的措施，但还有许多其他的措施。大多数优化只能应用于专门的情况，所以特定正则表达式只能从其中的某一些（甚至是没有）获益。

Perl 的调试模式（debugging mode）能提供优化的信息。在正则表达式第一次编译时，Perl 会选择这个正则表达式所使用的优化措施，而调试模式会显示其中的一部分。调试模式同样可以告诉我们引擎是如何应用表达式的。仔细分析这些调试信息不属于本书的范围，但我会在这里给出简要介绍。

在程序中可以通过 **`use re 'debug';`** 来显示调试信息，用 **`no re 'debug';`** 来关闭（上文曾出现过编译指示 `use re`，根据不同的参数，启用或禁用插值变量中的内嵌代码☐337）。

如果希望在整个脚本中启用此功能，可以使用命令行参数`-Mre=debug`。这很适合检查单个的正则表达式的编译方法。下面是一个例子（只保留了相关的行）：

```
❶ % perl -cw -Mre=debug -e 'm/^Subject: (.*)/'
❷ Compiling REx '^Subject: (.*)'
❸ rarest char j at 3
❹      1: BOL(2)
❺      2: EXACT <Subject: >(6)
            ......
❻     12: END(0)
❼ anchored 'Subject: ' at 0 (checking anchored) anchored(BOL) minlen 9
❽ Omitting $' $& $' support.
```

在❶处从 shell 提示符启动 *perl*，使用命令行参数**`-c`**（意思是检查脚本，而不是确切执行它），**`-w`**（如果 Perl 对代码存有疑问，就会发出警报），以及**`-Mre=debug`** 启用调试。**`-e`** 表示下面的参数‘`m/^Subject:·(.*)/`’是一段 Perl 代码，需要运行或者检查。

❸行报告表达式固定长度的字符串中“出现频率最低”的字符（由 Perl 猜测）。Perl 根据这一点进行某些优化（例如预查所需字符/子串☞245）。

第❶到❻行表示 Perl 编译好的正则表达式。因为篇幅的原因，我们在这里不会花太多的工夫。不过，即使是随便看看，第❺行也不难理解。

第❻行对应大多数行为。可能显示的信息包括：

`Anchored 'string' at offset`

　　它表示匹配必须包含某个字符串，此字符串在匹配中的偏移值为 *offset*。如果$紧跟在'*String*'之后，那么 *string* 是匹配结尾的元素。

`floating 'string' at from..to`

　　它表示匹配必须包含某个字符串，此字符串在匹配中处于从 from（开始）到 to（结束）中的任意位置。如果$紧跟在'*String*'之后，*string* 是匹配结尾元素。

`stclass 'list'`

　　它表示匹配可能的开始字符。

`anchored(MBOL), anchored(BOL), anchored(SBOL)`

　　说明表达式以「^」开头。MBOL 说明使用了/m 修饰符，BOL 和 SBOL 表示没有使用（BOL 和 SBOL 的区别在现代 Perl 中没有意义。SBOL 与$*变量有关，而此变量已被废弃了）。

`anchored(GPOS)`

　　说明正则表达式以「\G」开头。

`implicit`

　　说明 anchored(MBOL)是由 Perl 隐式添加的，因为正则表达式以「.*」开头。

`minlen length`

　　代表匹配成功的最小长度。

`with eval`

　　说明表达式包含「(?{…})」或是「(??{…})」。

第❼行与正则表达式无关，只有当二进制代码中的编译启用了-DDEBUGGING 时才会出现。如果启用，在载入整个程序之后，Perl 会报告是否启用了对$&等变量的支持（☞356）。

运行时调试信息

我们知道如何利用内嵌代码获得匹配的运行信息（☞331），但是 Perl 的正则表达式调试可以提供更多的信息。如果去掉表示“仅编译”的-c 选项，Perl 会提供更多关于匹配运行细节的信息。

出现“Match rejected by optimizer,”表示某种优化措施让传动装置认识到，这个正则表达式永远无法在目标字符串中匹配，所以会忽略任何尝试，下面是一个例子：

```
% perl -w -Mre=debug -e '"this is a test" =~ m/^Subject:/;'
    ......

Did not find anchored substr 'Subject:'?
Match rejected by optimizer
```

如果启用了调试功能，用户可以看到所有用到的正则表达式的调试信息，而不只限于用户提供的正则表达式。例如：

```
% perl -w -Mre=debug -e 'use warnings'
... 大量调试信息...
        ......
```

它没有进行任何操作，只是装载了 warning 模块，但是因为这个模块包含正则表达式，我们仍然会见到许多调试信息。

显示调试信息的其他办法

我已经提到，可以使用“use re 'debug';”或-Mre=debug 来启用正则表达式的调试。不过，如果把所有的 debug 替换为 debugcolor，而终端又支持 ANSI 转义控制字符（ANSI terminal control escape sequences），输出的信息就会以高亮标记，更容易阅读。

另一个办法是，如果 Perl 二进制代码在编译时启用了调试支持，可以使用命令行参数-Dr 来表示-Mre=debug。

结语

Final Comments

我确信自己已经陶醉于 Perl 的正则表达式中，本章的开头我曾提到，这是有充分理由的。Perl 之父 Larry Wall，完全是按照常识和发明的动力（Mother of Invention）来做的。是的，Perl 的正则表达式实现也有自己的问题，但是我仍然愿意醉心于 Perl 正则语言丰富的功能，及其与 Perl 其他部分的融合。

当然，我热情而不盲目——Perl 并没有提供某些我希望的特性。本书第 1 版渴望的某些特性现在已经添加了，我会继续提出要求，希望 Perl 会继续添加。相对于其他实现，Perl 最急需提供的功能是命名捕获（☞138）。本章给出了模拟的办法，但还存在若干限制。提供内建支持是最好的解决办法。如果能提供字符组集合运算（☞125）也很好，虽然目前可以费点周折用顺序环视来模拟（☞126）。

然后是占有优先量词（☞142）。Perl的固化分组提供了更多的完整功能，但是在某些情况下占有优先量词的解法更清楚更美观。所以，两种办法我都喜欢。事实上，还有两个我喜欢两个相关结构，目前还没有任何流派提供。其中之一是“cut”操作，或者叫「\v」，它会立刻清除当前存在的所有保存状态（这样，「x+\v」就等于「x++」或者「(?>x+)」）。另一个结构用来禁止传动装置的任何进一步的操作。它的意思是“要么在当前路径找到一个匹配，要么就不容许任何匹配，没有其他可能。”可能用「\V」来表示比较好。

还有个与「\V」有关的想法，我认为在传动装置中添加通用的钩子功能（general hooks）是有用的，这样第335页的程序就可以大大化简。

最后要说的是，我在第337页曾经提到，在内嵌代码插值到正则表达式时，提供更多的控制是非常有用的。

Perl当然不是理想的正则表达式语言，但它很接近这个目标，而且一直在进步。



第 8 章

Java

Java

自 2002 年早期发布的 Java 1.4.0 以后，Java 就内建了正则表达式包，`java.util.regex`，它的 API 毫不复杂（可以称得上简单），提供了强大而有创意的功能。对 Unicode 的支持很棒，文档很清晰，运行速度也很快。它能够用来匹配 `CharSequence` 对象，所以使用起来非常方便。

`sjava.util.regex` 一经发布就给人留下了深刻印象。它的功能、速度和正确性都达到了非常高的水平，尤其是对初次发布的软件来说，更是如此。Java 1.4 的最终版本是 1.4.2。写作本书时，Java 1.5.0（也叫 Java 5.0）已经发布，而 Java 1.6.0（也叫 Java 6.0）已经发布了第二个 beta 版本。本书针对的是 Java 1.5.0，不过我会在合适的时候提到它与 Java 1.4.2 或 Java 1.6.0 之间的重要差异（这些差异的总结在本章末尾☞401）（注 1）。

与之前各章的联系

在阅读本章之前，我必须说明，这一章不会重复第 1 章到第 6 章介绍的所有知识。有些只关心 Java 的读者可能会直接从这章开始阅读，我希望他们不要错过前言和开头几章的内容：第 1、2、3 章介绍了正则表达式的基本概念、特性和技巧，第 4、5、6 章包含了理解正则表达式的关键知识，它们可以直接应用到 `java.util.regex` 中。开头几章讲解的重要概念包括 NFA 引擎的工作原理、匹配优先性、回溯和效率。

注 1：本书可以适用于 Java 1.5.0 Update 7。Java 1.5 的最初版本在进行不区分大小写的匹配时存在若干 bug。Update2 修整了这些问题，所以如果你的 Java 版本低于它，我推荐你升级。Java 1.6 beta 的信息针对当前发布的 beta2, build59g 版。

表 8-1：方法名索引（按字母、页码排序）

380	appendReplacement	373	matcher	379	replaceFirst
381	appendTail	376	matcher(Matcher)	390	requireEnd
372	compile	395	matcher(Pattern)	392	reset
377	end	393	pattern(Matcher)	395	split
375	find	394	pattern(Pattern)	377	start
394	flags	395	quote	394	text
377	group	379	QuoteReplacement	377	toMatcheResult
377	groupCount	386	region	393	toString(Matcher)
388	hasAnchoringBounds	386	regionEnd	394	toString(Pattern)
387	hasTransparentBounds	386	regionStart	388	useAnchoringBounds
390	hitEnd	378	replaceAll	393	usePattern
376	lookingAt	382	replaceAllRegion	387	useTransparentBounds

这张表格供简要查询，详细的 API 讲解从第 371 页开始。

在这里我还是要强调，尽管第 367 页的表格查阅起来很方便，第 3 章第 114 和第 123 页的表格也是如此，但本书的目的不是作为参考手册，而是“掌握”正则表达式的详细教程。

前面几章已经出现过 java.util.regex 的例子（☞81、95、98、217、235），本章在讲解各种类及其实际应用时会给出更多的例子。不过，首先还是来看 Java 支持的正则流派，以及对应的修饰符。

Java 的正则流派

Java's Regex Flavor

java.util.regex 使用传统型 NFA，所以第 4、5、6 章介绍的丰富特性都适用于它。下页的表 8-2 总结了它的元字符。此流派的某些方面已经发生了变化，原因是各种匹配模式，匹配模式的启用通过各种 method 和 factory 来设定标志位，或者内嵌在表达式中的「(?*mods-mods*)」和「(?*mods-mods*:⋯)」修饰符。这些模式在第 368 页的表 8-3 中有列出。

下面是对表 8-2 的说明：

① 只有在字符组内部，\b 才代表退格字符。在其他场合，\b 都代表单词分界符。

表中给出的是“纯（raw）”反斜线，但是用作 Java 正则表达式的字符串时必须使用双反斜线。例如，表中的「\n」在 Java 的字符串中必须写作“\\n”。请参考“作为正则表达式的字符串”（☞101）。

\x##容许且只容许出现两位十六进制数字，所以「\xFCber」匹配‘über’。

表 8-2：java.util.regex 的正则流派

字符缩略表示法①	
☞115 (c)	\a [\b] \e \f \n \r \t \0*octal* \x*##* \u*####* \c*char*
字符组及相关结构	
☞118 (c)	字符组：[…] [^…]（可包含集合运算符☞125）
☞119	几乎任何字符：点号（根据模式的不同，有各种含义）
☞120 (c)	字符组缩略表示法②：\w \d \s \W \D \S
☞121 (c)	Unicode 属性和区块③：\p{*Prop*} \P{*Prop*}
锚点及其他零长度断言	
☞370	行/字符串起始位置：^ \A
☞370	行/字符串结束位置：$ \z \Z
☞370	当前匹配的起始位置：\G
☞133	单词分界符④：\b \B
☞133	环视结构⑤：(?=…) (?!…) (?<=…) (?<!…)
注释及模式修饰符	
☞135	模式修饰符：(?*mods*-*mods*) 容许出现的模式：x d s m i u
☞135	模式修饰范围：(?*mods*-*mods*:…)
☞368 (c)	注释：从#到行末（只在启用时有效）⑥
☞113 (c)	文字文本模式⑦：\Q…\E
分组及捕获	
☞137	捕获型括号：(…) \1 \2…
☞137	仅分组的括号：(?:…)
☞139	固化分组：(?>…)
☞139	多选结构：|
☞141	匹配优先量词：* + ? {n} {n,} {x,y}
☞141	忽略优先量词：*? +? ?? {n}? {n,}? {x,y}?
☞142	占有优先量词：*+ ++ ?+ {n}+ {n,}+ {x,y}+

(c) ——可用于字符组内部　　　　①……⑦见说明

\u####容许且只容许四位十六进制数字，例如，「**\u00FC**ber」匹配 'über'，「\u20AC」匹配 ' '。

\0*octal* 要求开头为 0，后接 1 到 3 位十进制数字。

\c*char* 是**区分大小写**的，直接对后面字符的十进制编码进行异或（xoring）操作。与我见过的任何流派都不一样，在这里\cA 和\ca 是不同的。\cA 等于传统意义上的\x01，\ca 则等价于\x21，匹配 '!'。

② **\w**、**\d** 和**\s**（以及对应的大写缩略法）只适用于 ASCII 字符，而不包括其他的字母、数字或者 Unicode 空白字符。也就是说，\d 等价于[0-9]，\w 等价于[0-9a-zA-Z]，\s 等价于[·\t\n\f\r\x0B]（\x0B 是 ASCII 中基本不用的 VT 字符）。

要覆盖完整的Unicode字符，可以使用Unicode属性（☞121）：用\p{L}表示\w，\p{Nd}表示\d，用\p{Z}表示\s。（把小写的p替换为大写的P，就可以对应\W、\D和\S）。

③ **\p{…}**和**\P{…}**支持 Unicode 属性和区块，以及某些额外的“Java 属性”。它不支持Unicode字母表。详细信息在下一页。

④ 对单词分界符元字符**\b**和**\B**来说，“单词字符”的规定不同于\w和\W。单词分界符能够识别Unicode字符，而\w和\W只能识别ASCII字符。

⑤ **顺序环视**结构中可以使用任意正则表达式，但是**逆序环视**中的子表达式只能匹配长度有限的文本。也就是说，「?」可以出现在逆序环视中，但「*」和「+」则不行。请参考第3章133页开始的内容。

⑥ 只有在使用/x修饰符，或者使用 Pattern.COMMENTS 选项（☞368）（请不要忘记在多行文本字符串中添加换行符，如第401页的例子）时，#…␤才算注释。没有转义的ASCII字空白字符会被忽略。**注意**：这一点与大多数支持此模式的正则引擎不同，在Java中字符组内部的注释和空白字符也会被忽略。

⑦ \Q…\E一直是被支持的，但在Java 1.6之前，字符组**内部**的此种结构是不可靠的。

表8-3：java.util.regex中Match和Regex的方法

编译选项	(?*mode*)	描述
Patern.UNIX_LINES	d	更改点号和「^」的匹配（☞370）
Pattern.DOTALL	s	点号能匹配任何字符（☞111）
Pattern.MULTILINE	m	扩展「^」和「$」的匹配规定（☞370）
Pattern.COMMENTS	x	宽松排列和注释模式（在字符组内部也有效）（☞72）
Pattern.CASE_INSENSITIVE	i	对ASCII字符进行不区分大小写的匹配
Pattern.UNICODE_CASE	u	对Unicode字符进行不区分大小写的匹配
Pattern.CANNON_EQ		Unicode“按规则等价(canonical equivalence)”匹配模式（不同编码中的同样字符视为相等☞108）
Pattern.LITERAL		将regex参数作为文字文本，而非正则表达式

Java 对\p{…}和\P{…}的支持

Java Support for \p{…} and \P{…}

「\p{…}」和「\P{…}」结构支持 Unicode 的属性和区块，也支持特殊的 "Java" 字符属性。这种支持针对 Unicode Version 4.0.0（Java 1.4.2 只支持 Unicode Version 3.0.0）。

Unicode 属性

Unicode 属性是通过\p{Lu}之类的缩写名字来引用的（参加 122 页的列表）。单字母属性名可以省略括号，\pL 等价于\p{L}。而\p{Lowercase_Letter}之类的长名称是不支持的。

Java 1.5 及之前的版本是不支持 **Pi** 和 **Pf** 属性的，因此，具有这种属性的字符不能用\p{P}来匹配（Java 1.6 支持）。

"未赋值的代码点" 属性\p{Cn}匹配的字符，不能由 "其他字符" 属性\p{C}来匹配。

Java 不支持组合属性\p{L&}。

Java 支持伪属性\p{all}，它等价于「(?S:.)」，但不支持\p{assigned}和\p{unassigned}伪属性，不过我们可以用\P{Cn}和\p{Cn}来代替。

Unicode 区块

Unicode block 的支持要求使用 'In' 前缀。请翻到第 402 页查阅具体的版本信息，了解「\p{…}」和「\P{…}」中能够出现的区块名称。

为了保持向后兼容性，Java 1.5 中有两个 Unicode 区块在 Unicode Version 3.0 和 4.0 之间发生了变化。除了 Unicode 4.0 提供的 **Combining Diacritical Marks for Symbols** 和 **Greek and Coptic** 之外，还可以使用本不属于 Unicode 4.0 的名称 **Combining Marks for Symbols** 和 **Greek**。

Java 1.4.2 有个涉及 **Arabic Presentation Forms-B** 和 **Latin Extended-B** 的 bug，在 Java 1.5 中已经修正。

特殊的 Java 字符属性

从 Java 1.5.0 开始，\p{…}和\P{…}结构能够支持 java.lang.Character 中未弃用（non-deprecated）的 *isSomething* 方法。为了在正则表达式中使用此功能，请把方法名开头的 'is' 替换成 'java'，然后使用「\p{…}」和「\P{…}」。例如，由 java.lang.Characer.isJavaIdentifierStart 匹配的字符也能用正则表达式「\p{javaJavaIdentifierStart}」来匹配（请参考 java.lang.Character 类的文档）。

Unicode 行终结符

Unicode Line Terminators

在 Unicode 之前的传统正则流派中，点号、^、$和\Z 会对换行符（ASCII 中的 LF 字符）进行特殊处理。在 Java 中，大多数 Unicode 行终结符（☞109）也会这样特殊处理。

正常情况下，Java 会把下面的这些字符当作行终结符：

字符代码	名称	说明
U+000A	LF \n	ASCII 换行符（“newline”）
U+000D	CR \r	ASCII 回车
U+000D U+000A	CR/LF \r\n	ASCII 回车/换行序列
U+0085	NEL	Unicode NEXT LINE（Unicode 换行符）
U+2028	LS	Unicode LINE SEPERATOR（Unicode 行分隔符）
U+2029	PS	Unicode PARAGRAPH SEPARATOR（Unicode 段分隔符）

根据匹配模式（☞368）的不同，点号、^、$和\Z 会对这些符号进行特殊处理：

匹配模式	受影响符号	说明
UNIX_LINE	^ . $ \Z	恢复到传统模式，只把换行符作为一行的终结
MULTILINE	^ $	字符串中的行终结符也可以由^和$匹配
DOTALL	.	行终结符不再另行处理，**点号**可以匹配所有字符

作为行终结标志的双字符 CR/LF 值得一提。默认情况（没有使用 UNIX_LINES 时）是识别完整的行终结符，匹配文本行边界的元字符会把 CR/LF 视为不可分隔的单位，一次性匹配这两个字符。

举例来说，$和\Z 通常会匹配行终结符之前的位置。LF 是行终结符，但只有在它不属于 CR/LF（也就是说，LF 之前**没有** CR）的情况下，$和\Z 才能匹配字符串末尾的 LF 之前的位置。

MUTILINE 模式中的$和^也是如此，在这种模式下，只有在 CR 之后没有 LF 的情况下，^才能匹配 CR 之后的位置；只有在 LF 之前不是 CR 的情况下，$才能匹配 LF 之前的位置。

必须说清楚的是，DOTALL 对 CR/LF 的处理没有影响（DOTALL 只影响**点号**，而点号总是逐个处理字符的），UNIX_LINES 根本不存在此类问题（它只识别 CR，所有其他行终结符都不需要特殊处理）。

使用 java.util.regex

Using java.util.regex

通过 java.util.regex 使用正则表达式非常简单，功能由两个类，一个接口和一个 unchecked exception 组成。

```
java.util.regex.Pattern
java.util.regex.Matcher
java.util.regex.MatchResult
java.util.regex.PatternSyntaxException
```

就我个人来说，更喜欢简称前两个为“pattern”和“matcher”，许多时候我们只会用到这两个类。简单地说，Pattern 对象就是编译好的正则表达式，可以应用于任意多个字符串，Matcher 对象则对应单独的实例，表示将正则表达式应用到某个具体的目标字符串上。

Java 1.5 新提供的 MatchResult，它封装了成功匹配的数据。匹配数据可以在下一次匹配尝试之前从 Matcher 本身获得，也可以提取出来作为 MatchResult 保存。

如果匹配尝试所使用的正则表达式格式不正确（例如，「[oops]」的语法就不正确），就会抛出 PatternSyntaxException 异常。这是一个 unchecked exception，继承自 java.lang.IllegalAgumentException。

下面是一个完整的、详细的程序，示范了简单的匹配：

```
public class SimpleRegexTest {
  public static void main(String[] args)
  {
     String myText = "this is my 1st test string";
     String myRegex = "\\d+\\w+"; // 表示\d+\w+
     java.util.regex.Pattern p = java.util.regex.Pattern.compile(myRegex);
     java.util.regex.Matcher m = p.matcher(myText);

     if (m.find()) {
        String matchedText = m.group();
        int matchedFrom = m.start();
        int matchedTo = m.end();
        System.out.println("matched [" + matchedText + "] " +
                           "from " + matchedFrom +
                           " to " + matchedTo + ".");
     } else {
        System.out.println("didn't match");
     }
  }
}
```

结果是‘**matched [1st] from 12 to 15.**’。在本章的所有例子中，我都用斜体表示变量名。如果这样声明，可以省略粗体部分：

```
import java.util.regex.*;
```

它应该放在程序的头部，和第 3 章的例子（☞95）一样。

这是标准的做法，而且程序更容易管理。本章的其他程序省略了 import 语句。

java.util.regex 使用的对象模型与其他大多数语言不同。请注意，前面例子中的 Matcher 对象 *m*，是通过把 Pattern 对象和目标字符串联系起来得到的，它用来进行实际的匹配尝试（使用 find 方法），以及查询结果（使用 group、start 和 end 方法）。

这办法初看起来可能有点古怪，但你很快就能适应了。

The Pattern.compile() Factory

The Pattern.compile() Factory

正则表达式的 Pattern 对象是通过 Pattern.compile 生成的。第一个参数是代表正则表达式的字符串（☞101）。368 页表格 8-3 中的选项可以作为第二个参数。下面的代码从字符串 *myRegex* 生成一个 pattern，进行不区分大小写的匹配：

```
Pattern pat = Pattern.compile(myRegex,
                  Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);
```

预定义的 pattern 常量用来指定编译选项（例如 Pattern.CASE_INSENSITIVE），这可能有点笨拙（注 2），所以我会使用正则表达式内部的模式修饰符（☞110）。包括 378 页的「(?x)」，399 页的「(?s)」和多个「(?i)」。

不过，预定义常量固然复杂，但这种“笨办法（unwieldy）”能够降低新手阅读代码的难度。如果没有页面宽度限制，我们可以这样写第 384 页的 Pattern.compile 的第二个参数：

```
Pattern.UNIX_LINES | Pattern.CASE_INSENSITIVE
```

而不是在正则表达式开头使用不那么清楚的「(?id)」。

从名字可以看出，这一步把正则表达式解析并编译为内部形式。第 6 章对此有详细讲解（☞241），简单地说，在字符串内应用表达式的整个过程中，编译 pattern 通常是最耗时间的。所以要把编译独立出来，作为第一步——这样就可以先期将正则表达式编译好，重复使用。

当然，如果正则表达式编译之后只需要使用一次，编译时机就不是个问题，但如果需要多次应用（例如应用到读入文件的每一行），预编译 Pattern 对象就很有价值。

注 2：尤其是在页面宽度固定的书籍中安排代码的时候——我对此深有体会。

调用 Patern.compile 可能抛出两种类型的异常：如果正则表达式不合规则，抛出 Pattern-SyntaxException，选项不合规则，抛出 IllegalArgumentException。

Pattern 的 matcher 方法

Pattern's matcher method

下一节（☞394）我们会看到，Pattern 提供了某些简便的方法，但是大多数情况下，我们只需要一个方法完成所有工作：matcher。它接受一个参数：需要检索的字符串（注 3）。此时并没有确切应用这个正则表达式，而只是为将 pattern 应用到特定的字符串做准备。matcher 方法返回一个 Matcher 对象。

Matcher 对象

The Matcher Object

把正则表达式和目标字符串联系起来，生成 Matcher 对象之后，就可以以多种方式将其应用到目标字符串中，并查询应用的结果。例如，对于给定的 Matcher 对象 m，我们可以用 m.find() 来把 *m* 的表达式应用到目标字符串中，返回一个 Boolean 值，表示是否能找到匹配。如果能找到，*m*.group()返回实际匹配的字符串。

在讲解 Matcher 的各种方法之前，不妨先了解了解它保存的各种信息。为了方便阅读，下面的清单都提供了对应的详细讲解部分的页码。第一张清单中的元素是程序员能够设置和更改的，而第二张清单中的元素是只读的。

程序员能够设置和修改的是：

- Pattern 对象，由程序员在创建 Matcher 时指定。可以通过 usePattern() 方法更改（☞393）。当前所用的 Pattern 可以用 pattern() 方法获得。
- 目标字符串（或其他 CharSequence 对象），由程序员在创建 Matcher 时指定。可以通过 reset(*text*) 方法更改（☞392）。
- 目标字符串的"**检索范围**（region）"（☞394）。默认情况下，检索范围就是整个目标字符串，但是程序员可以通过 region 方法，将其修改为目标字符串的某一段。这样某些（而不是全部）匹配操作就只能在某个区域内进行。

注 3：事实上，java.util.regex 是非常方便的，因为"string"参数可以是任何实现 Char-Sequence 接口的对象（String、StringBuffer、StringBuilder，以及 CharBuffer 都可以）。

当前检索范围的起始和结束偏移值可以通过 regionStart 和 regionEnd 方法获得（☞386）。reset 方法（☞392）会把检索范围重新设置为整个目标字符串，任何在内部调用 reset 方法的方法也是一样。

- *anchoring bounds* 标志位。如果检索范围不等于整个目标字符串，程序员可以设定，是否将检索范围的边界设置为“文本起始位置”和“文本结束位置”，这会影响文本行边界元字符（\A ^ $ \z \Z）。

  默认情况下，这个标志位为 true，但也可能改变，可以通过 useAnchoringBounds（☞388）和 hasAnchoringBounds 方法来修改和查询。Reset 方法不会修改标志位。

- *transparent bounds* 标志位。如果检索范围是整个目标字符串中的一段文本，设置为 true 容许各种“考察（looking）”结构（顺序环视、逆序环视，以及单词分界符）超越检索范围的边界，检查外部的文本。

  默认情况下，这个标志位为 false，但也可能改变，可以通过 useTransparentBounds（☞388）和 hasTran- sparentBounds 方法来修改和查询。Reset 方法不会修改标志位。

下面的只读数据保存在 matcher 内部：

- 当前 pattern 的捕获型括号的数目可以通过 groupCount（☞377）方法查询。
- 目标字符串中的 *match pointer* 或 *current location*，用于支持“寻找下一个匹配”的操作（通过 find 方法☞375）。
- 目标字符串中的 *append pointer*，在查找-替换操作中（☞380），复制未匹配的文本部分时使用。
- 表示到达字符串结尾的上一次匹配尝试是否成功的标志位。可以通过 hitEnd 方法（☞390）获得这个标志位的值。
- *match result*。如果最近一次匹配尝试成功，Java 会将各种数据收集起来，合称为 match result（☞376）。包括匹配文本的范围（通过 group() 方法），匹配文本在目标字符串中的起始和结束偏移值（通过 start() 和 end() 方法），以及每一组捕获型括号对应的信息（通过 group(*num*)、 start(*num*) 和 end(*num*) 方法）。

  match-result 数据封装在 MatchResult 对象中，通过 toMatchResult 方法获得。MatchResult 方法有自己的 group、start 和 end 方法（☞377）。

- 一个标志位，表明更长的目标文本是否会导致匹配失败（匹配成功之后才可用）。如果边界元字符影响了匹配结果的生成，则此值为 true。可以通过 `requireEnd` 方法查看它的值（☛390）。

上面列出的内容很多，但是如果按照功能分别讲解，便很容易掌握。这正是下面几节的内容。把这一章作为参考手册时，本章开头（☛366）的列表会很有用。

应用正则表达式

Applying the Regex

把 matcher 的正则表达式应用到目标文本时，主要会用到 `Matcher` 的这些方法：

`boolean` **`find()`**

此方法在目标字符串的当前检索范围（☛384）中应用 Matcher 的正则表达式，返回的 Boolean 值表示是否能找到匹配。如果多次调用，则每次都在上次的匹配位置之后尝试新的匹配。没有给定参数的 `find` 只使用当前的检索范围（☛384）。

下面是简单示例：

```
String regex = "\\w+"; // ⌈\w+⌋
String text  = "Mastering Regular Expressions";
Matcher m = Pattern.compile(regex).matcher(text);
if (m.find())
   System.out.println("match [" + m.group() + "]");
```

结果是：

```
match [Mastring]
```

如果这样写：

```
while (m.find())
   System.out.println("match [" + m.group() + "]");
```

结果就是：

```
match [Mastering]
match [Regular]
match [Expressions]
```

`boolean` **`find(int`** ***offset*****`)`**

如果指定了整型参数，匹配尝试会从距离目标字符串开头 *offset* 个字符的位置开始。如果 *offset* 为负数或超出了目标字符串的长度，会抛出 `IndexOutOfBoundsException` 异常。

这种形式的 find 不会受当前检索范围的影响，而会把它设置为整个“目标字符串”（它会在内部调用 reset 方法☞392）。

第 400 页的补充内容（作为第 399 页问题的答案）给出了恰当的例子。

boolean **matches()**

此方法返回的 Boolean 值表示 matcher 的正则表达式能否**完全**匹配目标字符串中当前检索范围的那段文本。也就是说，如果匹配成功，匹配的文本必须从检索范围的开头开始，到检索范围的结尾结束（默认情况就是整个目标字符串）。如果检索范围设置为默认的“所有文本”，matches 比使用「\A(?:…)\z」要简单。

不过，如果当前检索范围不等于默认情况（☞384），使用 matches 可以在不受 anchoring-bounds 标志位（☞388）影响的情况下检查整个检索范围中的文本。

举例来说，如果使用 CharBuffer 来保存程序中用户输入的文本，而检索范围设定为用户用鼠标选定的部分。如果用户点击选区的部分，可以用 *m*.usePattern(*urlPattern*).matches() 来检查选定部分是否为 URL（如果是，则进行相应的处理）。

String 对象也支持 matches 方法：

```
"1234".matches("\\d+"); // true
"123!".matches("\\d+"); // false
```

boolean **lookingAt()**

此方法返回的 Boolean 值表示 Matches 的正则表达式能否在当前目标字符串的当前检索范围中找到匹配。它类似于 matches 方法，但不要求检索范围中的整段文本都能匹配。

查询匹配结果

Querying Match Results

下面列出的 Matcher 方法返回了成功匹配的信息。如果正则表达式还未应用过，或者匹配尝试不成功，它们会抛出 IllegalStateException。接收 *num* 参数（对应一组捕获型括号）的方法，如果给定的 *num* 非法，会抛出 IndexOutOfBoundsException。

请注意 start 和 end 方法，它们返回的偏移值不受检索范围的影响——偏移值从整个目标字符串的开头开始计算，而不是检索范围的开头。

后面还给出了一个例子，讲解其中大部分方法的使用。

String **group()**

返回前一次应用正则表达式的匹配文本。

int **groupCount()**

返回正则表达式中与 Matcher 关联的捕获型括号的数目。在 group、start 和 end 方法中可以使用小于此数目的数字作为 *numth* 参数，下文介绍（注 4）。

String **gropu(**int *num***)**

返回编号为 *num*th 的捕获型括号匹配的内容，如果对应的捕获型括号没有参与匹配，则返回 null。如果 *num*th 为 0，表示返回整个匹配的内容，group(0)就等于 group()。

int **start(**int *num***)**

此方法返回编号为 *num*th 的捕获型括号所匹配文本的起点在目标字符串中的绝对偏移值——即从目标字符串起始位置开始计算的偏移值。如果捕获型括号没有参与匹配，则返回-1。

int **start()**

此方法返回整个匹配起点的绝对偏移值，start()就等于 start(0)。

int **end(**int *num***)**

此方法返回编号为 *num*th 的捕获型括号所匹配文本的终点在目标字符串中的绝对偏移值——即从目标字符串起始位置开始计算的偏移值。如果捕获型括号没有参与匹配，则返回-1。

int **end()**

次方法返回整个匹配的终点的绝对偏移值，end()就等于 end(0)。

MatcheResult **toMatchResult()**

此方法从 Java 1.5.0 开始提供，返回的 MatchResult 对象封装了当前匹配的信息。它和 Matcher 类一样，也包含上面列出的 group、start、end 和 groupCount 方法。

如果前一次匹配不成功，或者 Matcher 还没有进行匹配操作，调用 toMatcheResult 会抛出 IllegalStateException。

注 4：groupCount 方法任何时候都可调用，而这里列出的其他方法必须在匹配尝试成功之后才可调用。

示例

下面的例子示范了若干方法的使用。给定一个 URL 字符串，这段代码会找出 URL 的协议名（‘http’ 或是 ‘https’）、主机名，以及可能出现的端口号：

```
String url   = "http://regex.info/blog";
String regex = "(?x) ^(https?):// ([^/:]+) (?:(\\d+))?";
Matcher m = Pattern.compile(regex).matcher(url);

if (m.find())
{
  System.out.print(
      "Overall  [" + m.group()  + "]" +
      " (from "    + m.start()  + " to " + m.end()  + ")\n" +
      "Protocol [" + m.group(1) + "]" +
      " (from "    + m.start(1) + " to " + m.end(1) + ")\n" +
      "Hostname [" + m.group(2) + "]" +
      " (from "    + m.start(2) + " to " + m.end(2) + ")\n"
  );
  // group(3)可能未参与匹配，此处应小心对待
  if (m.group(3) == null)
     System.out.println("No port; default of '80' is assumed");
  else  {
     System.out.print("Port is [" + m.group(3) + "] " +
                      "(from " + m.start(3) + " to " + m.end(3) + ")\n");
  }
}
```

执行的结果是：

```
Overall  [http://regex.info] (from 0 to 17)
Protocol [http] (from 0 to 4)
Hostname [regex.info] (from 7 to 17)
No port; default of '80' is assumed
```

简单查找-替换

Simple Search and Replace

上面介绍的方法足够进行查找-替换操作了，只是比较麻烦，但是 Matcher 提供了简便的方法。

`String` **`replaceAll`**`(String` *`replacement`*`)`

返回目标字符串的副本，其中 Matcher 能够匹配的文本都会被替换为 *replacement*，具体处理过程在 380 页。

此方法不受检索范围的影响（它会在内部调用 `reset` 方法），不过第 382 页介绍了在检索范围中进行这样操作的方法。

`String` 类也提供了 `replaceAll` 的方法，所以：

```
string.replaceAll(regex, replacement)
```

就等于：

```
Pattern.compile(regex).matcher(string).replaceAll(replacement)
```

```
String replaceFirst(String replacement)
```

此方法类似 replaceAll，但它只对第一次匹配（如果存在）进行替换。

String 类也提供了 replaceFirst 方法。

```
static String quoteReplacement(String text)
```

此 static 方法从 Java 1.5 开始提供，返回 *text* 的文字用作 *replacement* 的参数。它为 text 副本中的特殊字符添加转义，避免了下一页讲解的正则表达式特殊字符处理（下一节也给出了 Matcher.quoteReplacement 的例子）。

简单查找-替换的例子

下面的程序将所有的“Java 1.5”改为“Java 5.0”，用市场化的名称取代开发名称：

```
String text = "Before Java 1.5 was Java 1.4.2. After Java 1.5 is Java 1.6";
String regex = "\\bJava\\s*1\\.5\\b";
Matcher m = Pattern.compile(regex).matcher(text);
String result = m.replaceAll("Java 5.0");
System.out.println(result);
```

结果是：

```
Before Java 5.0 was Java 1.4.2. After Java 5.0 is Java 1.6
```

如果 pattern 和 matcher 不需要复用，可以使用链式编程：

```
Pattern.compile("\\bJava\\s*1\\.5\\b").matcher(text).replaceAll("Java 5.0")
```

（如果单个线程中需要多次用到同一 pattern，预编译 pattern 对象可以提升效率☞372）

对正则表达式稍加修改，就可以用它来把“Java 1.6”改为“Java 6.0”（当然也需要修改 replacement 字符串，讲解见下页）。

```
Pattern.compile("\\bJava\\s*1\\.([56])\\b").matcher(text).replaceAll("Java $1.0")
```

对于同样的输入文本，结果如下：

```
Before Java 5.0 was Java 1.4.2. After Java 5.0 is Java 6.0
```

如果只需要替换第一次出现的文本，可以使用 replaceFirst，而不是 replaceAll。除了这种情况，还有一种情况可以使用 replaceFirst，即明确知道目标字符串中只存在一个匹配时，使用 replaceFirst 可以提高效率（如果了解正则表达式或是目标字符串，可以预知这一点）。

replacement 参数

replaceAll 和 replaceFirst 方法（以及下一节的 appendReplacement 方法）接收的 *replacement* 参数在插入到匹配结果之前，会进行特殊处理：

- '$1'、'$2' 之类会替换为对应编号的捕获型括号匹配的文本（$0 会被替换为所有匹配的文本）。
- 如果 '$' 之后出现的不是 ASCII 的数字，会抛出 IllegalArgumentException 异常。

  '$' 之后的数字，只会应用"有意义"的部分。如果只有 3 组捕获型括号，则 '$25' 会被视为$2 然后是 '5'，而此时$6 会抛出 IndexOutOfBoundsException。
- 反斜线用来转义后面的字符，所以 '\$' 表示美元符号。同样的道理，'\\' 表示反斜线（在 Java 的字符串中，表示正则表达式中的 '\\' 需要用四个斜线 '\\\\'）。同样，如果有 12 组捕获型括号，而我们希望使用第一组捕获型括号匹配的文本，然后是 '2'，应该是这样 '$1\2'。

如果不清楚 replacement 字符串的内容，最好使用 Matcher.quoteReplacement 方法，确保不会出错。如果用户的正则表达式是 *uRegex*，replacement 是 *uRepl*，下面的做法可以确保替换的安全：

```
Pattern.compile(uRegex).matcher(text).replaceAll(Matcher.quoteReplacement(uRepl))
```

高级查找-替换

Advanced Search and Replace

有两个方法可以直接操作 Matcher 的查找-替换过程。它们配合起来，把结果存入用户指定的 StringBuffer 中。第一种方法每次匹配之后都会调用，在 result 中存入 replacement 字符串和匹配之间的文本。第二种方法会在所有匹配完成之后调用，将目标字符串中最后的文本拷贝过来。

Matcher **appendReplacement**(StringBuffer *result*, String *replacement*)

在正则表达式应用成功之后（通常是 find）马上调用此方法会把两个字符串添加到指定的 *result* 中：第一个是原始目标字符串匹配之前的文本，然后是经过上面讲解的特殊处理的 *replacement* 字符串。

举例来说，如果 matcher *m* 与正则表达式「\w+」和‘-->one+test<--’相联系，while 循环的第一轮情况如下：

```
while (m.find())
    m.appendReplacement(sb, "XXX")
```

find 找到下画线标注的部分‘-->one+test<--’。

然后，第一次 appendReplacement 调用会在 StringBuffer *result* 中加入匹配之前的文本‘-->’，跳过匹配的文本，再插入 replacement 字符串‘XXX’。

While 循环的第二轮，find 匹配‘-->one+test<--’。此时调用 appendReplacement 会给 sb 附加上匹配之前的文本‘+’，然后仍然是字符串‘XXX’。

这样 sb 的值就是‘-->XXX+XXX’，m 在原始目标字符串中对应的位置是‘-->one+test<--’。现在该使用 appendTail 方法了，下文将会介绍。

StringBuffer **appendTail**(StringBuffer *result*)

找到所有匹配之后（或者是，找到期望的匹配之后——此时用户可以停止匹配过程），这个方法将目标字符串中剩下的文本附加到提供的 StringBuffer 中。

在上例中，接下来就是：

```
m.appendTail(sb)
```

将‘<--’附加到 sb。这样就得到‘-->XXX+XXX<--’，查找-替换完成。

查找-替换示范

下面实现了一个自己的 replaceAll 的方法（并非必须，只是作为示范）。

```
public static String replaceAll(Matcher m, String replacement)
{
    m.reset(); // 保证 Matcher 不受之前的影响
    StringBuffer result = new StringBuffer(); // 生成用于替换的副本
    while (m.find())
        m.appendReplacement(result, replacement);
    m.appendTail(result);
   return result.toString(); // 转换为 String，然后返回
}
```

它与 Java 内建的 replaceAll 方法一样，它不受检索范围（☞384）的影响，而是在查找-替换之前重置检索范围。

为了弥补这个缺憾，下面的 replaceAll 只在检索范围中进行，修改和新增的代码会标注出来：

```
public static String replaceAllRegion(Matcher m, String replacement)
{
   Integer start = m.regionStart();
   Integer end = m.regionEnd();
   m.reset().region(start, end); // 重置 matcher，之后恢复 region
   StringBuffer result = new StringBuffer(); //生成用于替换的副本

   while (m.find())
      m.appendReplacement(result, replacement);

    m.appendTail(result);
    return result.toString(); //转换为 String，然后返回
}
```

这里使用“**方法链**（译注 1）”结合 reset 和 region，详细讲解请参阅第 389 页。

下面的程序更加完善，它将 metric 变量中的摄氏温度转换为华氏温度：

```
// 构建一个 matcher，匹配"Metric"变量中后面跟有"C"的数值
// 下面的正则表达式是：「( \d+(?:\.\d* )? )C\b」
Matcher m = Pattern.compile("(\\d+(?:\\.\\d*)?)C\\b").matcher(metric);
StringBuffer result = new StringBuffer(); //生成用于替换的副本
while (m.find())
{
  float celsius = Float.parseFloat(m.group(1));   // 得到数值，转换为浮点数
  int fahrenheit = (int) (celsius * 9/5 + 32);   // 转换为华氏温度
  m.appendReplacement(result, fahrenheit + "F"); // 插入
}
m.appendTail(result);
System.out.println(result.toString()); // 显示结果
```

如果 metric 变量包含“**from 36.3C to 40.1C.**”，结果就是“**from 97F to 104F.**”。

原地查找-替换

In-Place Search and Replace

现在还只出现过对 String 对象使用 java.util.regex 的例子，但是 Matcher 其实适用于任何实现了 CharSequence 接口的类，所以我们能够对目标文本进行实时的、原地（in place）的修改。

StringBuffer 和 StringBuilder 是两种常用的实现了 CharSequence 接口的类，前者保证线程安全性，但效率略低。这两者都可以作为 String 来使用，但它们的值可以变化。本书中的例子使用了 StringBuilder，但如果在多线程环境中，请使用 StringBuffer。

译注 1：method chaining，也叫“链式编程”。

这个简单的例子在 StringBuilder 中搜索大写单词，将它们替换为小写形式（注 5）：

```
StringBuilder text = new StringBuilder("It's SO very RUDE to shout!");
Matcher m = Pattern.compile("\\b[\\p{Lu}\\p{Lt}]+\\b").matcher(text);
while (m.find())
   text.replace(m.start(), m.end(), m.group().toLowerCase());
System.out.println(text);
```

结果是：

```
It's so very rude to shout!
```

其中匹配了两次，调用了两次 text.replace。头两个参数指定需要替换的文本范围（跳过表达式匹配的文本），然后是用作 replacement 的文本（也就是匹配文本的小写形式）。

因为 replacement 和匹配文本长度相同，原地的查找-替换很简单。如果不需要迭代进行查找-替换，这种方法非常方便。

长度变化的替换

如果 replacement 的长度不同于要替换文本的长度，情况就复杂起来。对目标字符串的修改是在“背后”进行的，所以“**匹配指针**（match pointer）”（在目标字符串中进行下一次 find 的开始位置）会发生错误。

要解决这个问题，我们可以自己维护匹配指针，明确告诉 find 下一次尝试应该从何处开始。下面的例子做了这样的改进，在需要插入的小写文本两端添加了…：

```
StringBuilder text = new StringBuilder("It's SO very RUDE to shout!");
Matcher m = Pattern.compile("\\b[\\p{Lu}\\p{Lt}]+\\b").matcher(text);
int matchPointer = 0;// 首先从字符串起始位置开始
while (m.find(matchPointer)) {
   matchPointer = m.end(); // 记录本次匹配结束的位置
   text.replace(m.start(), m.end(), "<b>"+ m.group().toLowerCase() +"</b>);
   matchPointer += 7; //算上添加的'<b>'和'</b>'
}
System.out.println(text);
```

结果是：

```
It's <b>so</b> very <b>rude</b> to shout!
```

注 5：代码中的表达式是「\b[\p{Lu}\p{Lt}]+\b」。在第 3 章（☞123）介绍过，\p{Lu} 匹配 Unicode 中的所有大写字母，\p{Lt} 匹配首字母形式字符。对应 ASCII 编码的正则表达式是「\b[A-Z]+\b」。

Matcher 的检索范围

The Matcher's Region

从 Java1.5 开始，Matcher 支持可变化的检索范围，通过它，我们可以把匹配尝试限制在目标字符串之中的某个范围。通常情况下，Matcher 的检索范围包含整个目标字符串，但可以通过 `region` 方法实时修改。

下面的例子检索 HTML 代码字符串，报告不包含 ALT 属性的 image tag。对同一段文本（HTML），它使用了两个 Matcher：一个寻找 image tag，另一个寻找 ALT 属性。

尽管两个 Matcher 应用到同一个字符串，但它们的关联只局限于，用找到的 image tag 来限定寻找 ALT 属性的范围。在调用 ALT-matcher 的 `find` 方法之前，我们用刚刚完成的 image-tag 的匹配来设定 ALT-matcher 的检索范围。

单拿出 image tag 的 body 之后，就可以通过查找 ALT 来判断当前的 tag 内是否包含 ALT 属性：

```
// 查找 image tag 的 matcher。变量'html'包含需要处理的 HTMl 代码
Matcher mImg = Pattern.compile("(?id)<IMG\\s+(.*?)/?>").matcher(html);

// 查找 ALT 属性的 matcher（应用于刚刚找到的 IMG tag 中）
Matcher mAlt = Pattern.compile("(?ix)\\b ALT \\s* =").matcher(html);

// 对 html 中的每个 image tag
while (mImg.find()) {
   // 把查找范围局限在刚刚找到的 tag 中
   mAlt.region(mImg.start(1), mImg.end(1) );

   // 如果没有找到，则报错，输出找到的整个 image tag
   if (! mAlt.find())
      System.out.println("Missing ALT attribute in: " + mImg.group());
}
```

或许在一处指定目标字符串（创建 *`mAlt`* Matcher 时），另一处指定检索范围（调用 *`mAlt`*`.region` 时）的做法有些怪异。果真如此的话，我们可以为 *`mAlt`* 创建虚构的目标字符串（一个空字符串），然后每次都调用 *`mAlt`*`.`**`reset`**`(`*`html`*`).region(…)`。调用 `reset` 可能会降低些效率，但是同时设定目标字符串和检索范围的逻辑更加清晰。

无论采取哪种办法，都必须明白，如果不设定 ALT Matcher 的检索范围，对它调用 `find` 就会检索整个目标字符串，返回无关的 '`ALT=`' 属性。

下面继续完善这个程序，返回找到的 image tag 在 HTML 代码中的行数。我们首先隔离出 image tag 之前的 HTML 代码，然后计算其中的换行符数。

标注部分为新增代码：

```
// 查找 image tag 的 matcher。 变量'html'包含需要处理的 HTML 代码
Matcher mImg = Pattern.compile("(?id)<IMG\\s+(.*?)/?>").matcher(html);
// 查找 ALT 属性的 matcher (应用于刚刚找到的 IMG tag 中)
Matcher mAlt = Pattern.compile("(?ix)\\b ALT \\s* =").matcher(html);
// 查找换行符的 Matcher
Matcher mLine = Pattern.compile("\\n").matcher(html);
// 对 HTML 中的每个 img tag ...
while (mImg.find())
{
   // 把查找范围局限在刚刚找到的 tag 中
   mAlt.region(mImg.start(1), mImg.end(1) );
   // 如果没有找到，则报错，输出找到的整个 image tag
   if (! mAlt.find()) {
      // 计算当前 image tag 之前的换行符的数量
      mLine.region(0, mImg.start());
      int lineNum = 1; // 第一行编号为 1
      while (mLine.find())
            lineNum++; // 每遇到一个换行符就加 1
      System.out.println("Missing ALT attribute on line " + lineNum);
   }
}
```

与之前一样，每次设定 ALT Matcher 的检索范围时，都使用 image matcher 的 `start(1)` 方法得到 image tag *body* 在 HTML 中的起始位置。相反，在设定换行符匹配的检索范围终点时，使用 `start()` 方法来判断整个 *image tag* 的开始位置（也就是换行符计算的终点）。

几点提醒

记住，某些检索相关的方法并非不受检索范围的影响，而是它们在内部调用了 `reset` 方法，把检索范围设定为默认的“全部文本”。

- 受检索范围影响的查找方法：

```
matches
lookingAt
find() (不带参数)
```

- 会重置 matcher 及其检索范围的方法：

```
find(text) (带一个参数)
replaceAll
replaceFirst
reset (无须多言)
```

另外请记住，匹配结果数据中的偏移值（也就是 `start` 和 `end` 方法返回的数值）是不受检索范围影响的，它们只与整个目标字符串的开始位置有关。

设定及查看检索范围的边界

与设定及查看检索范围边界的方法有 3 个：

`Matcher region(int start, int end)`

将 matcher 的检索范围设定在整个目标字符串的 *start* 和 *end* 之间，数值均从目标字符串的起始位置开始计算。它同样会重置 Matcher，将 **"匹配指针"** 指向检索范围的开头，所以下一次调用 `find` 从此处开始。

如果没有重新设定，或是调用 `reset` 方法（无论是显式调用还是在其他方法内部调用 ☞392），检索范围都不会变化。

返回值为 Matcher 对象本身，所以此方法可用于方法链（☞389）。

如果 *start* 或 *end* 超出了目标字符串的范围，或者 *start* 比 *end* 要大，会抛出 `IndexOutOfBoundsException`。

`int regionStart()`

返回当前检索范围的起始位置在目标字符串中的偏移值，默认为 0。

`int regionEnd()`

返回当前检索范围的结束位置在目标字符串中的偏移值，默认为目标字符串的长度。

因为 `region` 方法要求同时提供 *start* 和 *end*，如果只希望设置其中一个，可能不太方便操作。表 8-4 给出了方法。

表 8-4：设定检索范围的单个边界

起始边界	结束边界	Java 代码
明确设定	不修改	`m.region(start, m.regionEnd());`
不修改	明确设定	`m.region(m.regionStart(), end);`
明确设定	不修改	`m.reset().region(start, m.regionEnd());`
不修改	明确设定	`m.region(0, end);`

超越检索范围

如果将检索范围设定为整个目标字符串中的一段，正则引擎会忽略范围之外的文本。也就是说，检索范围的起始位置可以用「^」匹配，而它可能并不是目标字符串的起始位置。

不过，某些情况下也可以检查检索范围之外的文本。启用 *transparent bounds* 能够让"考察

（looking）”结构检查范围之外的文本，如果禁用 *anchoring bounds*，则检索范围的边界不会被视为输入数据的起始/结束位置（除非实际情况如此）。

修改这两个标志位的理由与修改默认检索范围密切相关。之前用到了检索范围的例子都不需要设定这两个标志位——与检索范围相关的查找既不需要锚点也不需要“考察”结构。

如果程序把需要用户编辑的文本存放在 `CharBuffer` 中，用户希望执行查找-替换操作，就应当把操作的范围限定在光标之后的文本中，所以应当把检索范围的起始位置设定为当前光标所在的位置。如果用户的光标指向下面的位置：

```
Madagas‸car is much too large to see on foot, so you'll need a car.
```

要求把「`\bcar\b`」替换为“automobile”。设定了检索范围之后（即将其设定为光标之后的文本），你可能会很惊奇地发现第一次匹配就是在检索范围的开头：‘`Madagascar`’。这是因为默认情况下 transparent bounds 标志位设定为 false，因此「`\b`」将检索范围起始位置设定为文本的起始位置，而“看不到”左侧还有字符。如果将 transparent bounds 设定为 true，「`\b`」就能看到‘`c`’之前还有‘`s`’，因此「`\b`」不能匹配。

Transparent Bounds

与这个标志位相关的方法有两个：

`Matcher` **`useTransparentBounds`**`(boolean` *`b`*`)`

设定 transparent bounds 的值。默认为 *false*。

此方法返回 Matcher 本身，故可用在方法链中。

`boolean` **`hasTransparentBounds()`**

如果 transparent 生效，则返回 *true*。

Matcher 的 transparent-bounds 默认为 *false*。也就是说检索范围的边界在顺序环视、逆序环视和单词分界符“考察”时是不透明的。这样，正则引擎不会感知到检索边界之外的字符（注 6）。

注 6：Java 1.5 Update 7 中有个 bug，我已经报告给 Sun。使用 `Pattern.MULTILINE` 之后，如果之前正好有一个行终结符，即使已经取消了 anchoring bounds，「`^`」（如果修改了检索边界，它可以看作某种查看结构）也可以匹配检索范围的起始位置。

也就是说尽管检索范围的起始位置可能在某个单词内部，「\b」仍然能够匹配——它看不到之前的字母。

下面的例子说明了 transparent bounds 设置为 *false*（默认）的情况：

```
String regex = "\\bcar\\b"; // 「\b car\b」
String text  = "Madagascar is best seen by car or bike.";
Matcher m = Pattern.compile(regex).matcher(text);
m.region(7, text.length());
m.find();
System.out.println("Matches starting at character " + m.start());
```

结果是：也就是说尽管检索范围的起始位置可能在某个单词内部，「\b」仍然能够匹配——它看不到之前的字母。

```
Matches starting at character 7
```

单词分界符的确匹配了检索范围的起始位置，即 Madagas‸car，尽管此处根本不是单词的边界。如果不设定 transparent bounds，单词分界符就“受骗（spoofed)”了。

如果在 find 之前添加这条语句：

```
m.useTransparentBounds(true);
```

结果就是：

```
Matches starting at character 27
```

因为边界现在是透明的，引擎可以感知到起始边界之前有个字母‘s’，所以「\b」在此处无法匹配。于是结果就成了“…by·car·or·bike.”。

同样，transparent-bounds 只有在检索范围不等于“整个目标字符串”时才有意义。即使 reset 方法也不会重置它。

Anchoring bounds

与 anchoring bounds 有关的方法有：

Matcher **useAnchoringBounds**(Boolean *b*)

设置 matcher 的 anchoring-bounds 的值，默认为 *true*。

此方法会返回 matcher 对象本身，故可用于方法链中。

boolean **hasAnchoringBounds**()

如果启用了 anchoring bounds，则返回 *true*，否则返回 *false*。

默认状态下，anchoring bounds 为 *true*，也就是说行锚点（^ \A $ \z \Z）能匹配检索范围的边界，即检索范围不等于整个目标字符串。将它们设置为 *false* 表示行锚点只能匹配检索

范围内，整个目标字符串中符合规定的位置。

禁用 anchoring bounds 的理由可能与使用 transparent bounds 一样，当用户的“光标不在整段文本的起始位置时”保证语意的完备。

与 transparent-bounds 一样，anchoring bounds 也只有在检索范围不等于“整个目标字符串”时才有意义。即使 reset 方法也不会重置它。

方法链

Method Chaining

下面的程序初始化一个 Matcher，并设定某些选项：

```
Pattern p = Pattern.compile(regex);    // 编译 regex
Matcher m = p.matcher(text);           // 将 regex 与 text 建立联系,创建 Matcher
m.region(5, text.length());            // 将检索范围设定为从第 5 个字符开始
m.useAnchoringBounds(false);           // ^之类不能匹配检索范围的起始位置
m.useTransparentBounds(true);          // 考察结构能够超越检索范围
```

在前面的例子中我们看到，如果创建 Matcher 之后不再需要 regex，可以把前面两步合并起来：

```
Matcher m = Pattern.compile(regex).matcher(text);
m.region(5, text.length());           // 将检索范围设定为从第 5 个字符开始
m.useAnchoringBounds(false);          // ^之类不能匹配检索范围的起始位置
m.useTransparentBounds(true)          // 考察结构能够超越检索范围
```

不过，因为 Matcher 的两个方法会返回 Matcher 本身，可以把它们整合成一行（尽管因为排版的原因必须列为两行）：

```
Matcher m = Pattern.compile(regex).matcher(text).region(5, text.length())
                .useAnchoringBounds(false).useTransparentBounds(true);
```

功能并没有增加，但用起来更加简便。这种“方法链”格式紧凑，一行程序可能很难对应到单个步骤的文档，不过，好的文档重在说明“为什么”而不是“干什么”，所以这可能并不是个问题。在第 399 页的程序中，使用方法链可以保证格式紧凑清晰。

构建扫描程序

Methods for Building a Scanner

Java 1.5 的 matcher 提供了两个新方法，hitEnd 和 requireEnd，它们主要用来构建扫描程序（Scanner）。扫描程序将字符流解析为记号（token）流。举例来说，编译器的扫描程序会把‘var·<·34’解析为三个记号：INDENTIFIER·LESS_THAN·INTEGER。

这两个方法帮助扫描程序决定，刚刚完成的匹配尝试的结果是否应该用来解释（interpretation）当前输入。一般来说，只要其中一个返回 true，就表示在做出决定之前还应该输入更多的数据。例如，如果当前的输入数据（比如用户在交互式调试器中输入的命令）是单个字符‘<’，最好还是看看下一个字符是否‘=’，才能决定这个记号是 LESS_THAN 还是 LESS_THAN_OR_EQUAL。

在大多数应用正则表达式的项目中，这两个方法可能派不上用场，可是一旦需要，它们就是不可替代的。在这些不常见的场合，Java 1.5 中 hitEnd 方法存在的 bug 就很让人恼火。不过，看起来 Java 1.6 已经修正了这个错误，Java 1.5 中的解决办法将在本章末尾介绍。

构建扫描程序已经远远偏离了本书的主旨，所以我只会介绍些具体方法的定义，并给出例子（如果你确实需要扫描程序，应该去看看 java.util.Scanner）。

boolean **hitEnd()**

（Java 1.5 中这个方法是不可靠的，解决办法参见第 392 页）。

此方法表示，正则引擎在上一次匹配尝试中，是否检查了输入数据结束之后的数据（而无论上一次匹配是否成功）。其中包含「\b」和「$」之类的边界检查。

如果 hitEnd 返回 true，则输入更多数据可能会改变匹配结果（匹配成功变为匹配失败，匹配失败变为匹配成功，或者匹配文本发生变化）。相反，如果返回 false，则匹配结果已经确定，输入更多的数据也不会改变匹配结果。

常见的应用是，如果匹配成功，而 hitEnd 返回 true，则必须等待更多的输入数据才能最后做出决定。如果匹配失败，而 hitEnd 返回 false，应该期待更多的输入数据，而不是报告语法错误。

boolean **requireEnd()**

此方法只有在匹配成功之后才有意义，它表示正则引擎的匹配成功与否是否受输入数据结尾的影响。也就是说，如果 requireEnd 返回 true，更多的输入数据可能导致匹配

尝试失败，如果返回 false，更多的输入数据可能改变匹配的细节，但不会导致匹配失败。

常见的应用是，如果 `requireEnd` 返回 true，在最后做出决定之前，必须接受更多的输入数据。

`hitEnd` 和 `requireEnd` 都受到检索范围的影响。

使用 hitEnd 和 requireEnd 的例子

表 8-5 给出了在 `lookingAt` 搜索之后使用 `hitEnd` 和 `requireEnd` 的例子。所给的两个例子虽然很简单，但足够解释这两个方法。

表 8-5：在 lookingAt 搜索之后使用 hitEnd 和 requireEnd 的例子

	正则表达式	目标文本	匹配文本	hitEnd()	requierEnd()
1	`\d+\b|[><]=?`	'1234'	'1234'	true	true
2	`\d+\b|[><]=?`	'1234·>·567'	'1234·>·567'	false	false
3	`\d+\b|[><]=?`	'>'	'>'	true	false
4	`\d+\b|[><]=?`	'>·567'	'>·567'	false	false
5	`\d+\b|[><]=?`	'>='	'>='	false	false
6	`\d+\b|[><]=?`	'>=·567'	'>=·567'	false	false
7	`\d+\b|[><]=?`	'oops'	匹配失败	false	
8	`(set|setup)\b`	'se'	匹配失败	true	
9	`(set|setup)\b`	'set'	'set'	true	true
10	`(set|setup)\b`	'setu'	匹配失败	true	
11	`(set|setup)\b`	'setup'	'setup'	true	true
12	`(set|setup)\b`	'set·x=3'	'set·x=3'	false	false
13	`(set|setup)\b`	'setup·x'	'setup·x'	false	false
14	`(set|setup)\b`	'self'	匹配失败	false	
15	`(set|setup)\b`	'oops'	匹配失败	false	

表 8-5 中上面 7 行的正则表达式寻找一个非负整数以及 4 个比较运算符：大于、大于等于、小于、小于等于。下面 8 行的正则表达式更简单，寻找单词 `set` 或是 `setup`。这些例子很简单，但能说明问题。

举例来说，第 5 行中，虽然整个目标字符串都能匹配，`hitEnd` 仍然会返回 false。原因在于，尽管匹配文本包含目标字符串的最后一个字符，引擎也不需要检查之后的字符（无论是字符还是分界符）。

hitEnd 的 bug 及解决办法

Java 1.5 中的 hitEnd 方法存在 bug（Java 1.6 已经修正）（注 7），在某些特殊情况下 hitEnd 会得到不可靠的结果：在不区分大小写的匹配模式下，如果正则表达式的某个可选元素为单个字符（尤其是当它的匹配尝试失败时），就会出错。

例如，在不区分大小写的情况下使用「>=?」（它作为大的正则表达式的一部分）会诱发这个错误，因为‘=’是可选的单个字符。在不区分大小写的情况下使用「a|an|the」（仍然是包含在大的正则表达式中）也会诱发这个错误，因为单个字符「a」是众多多选分支之一，因此是可选的。

另两个例子是「values?」和「\r?\n\r?\n」。

解决办法 解决的办法是破坏诱发条件，或者禁用不区分大小写的匹配（至少是对诱发的子表达式禁用），或者是把单个字符替换为其他元素，例如字符组。

第一种办法会把「>=?」替换为「(?-i:>=?)」，使用模式修饰范围（☞110）保证不区分大小写的匹配不会应用于这个子表达式（这里不存在大小写的区别，所以这种办法完全没问题）。

如果使用第二种办法，「**a**|an|the」就变成了「**[aA]**|an|the」，代表了使用 Pattern.CASE_INSENSITIVE 进行不区分大小写匹配的情况。

Matcher 的其他方法

Other Matcher Methods

这些 Matcher 方法尚未介绍过：

Matcher **reset()**

这个方法会重新初始化 Matcher 的大多数信息，弃用前一次成功匹配的所有信息，将匹配位置指向文本的开头，把检索范围（☞384）恢复为默认的“全部文本”。只有 anchoring bounds 和 transparent bounds（☞388）不会变化。

Matcher 有三个方法会在内部调用 reset，因此也会重新设定检索范围：replaceAll、replaceFirst，以及只使用一个参数的 find。

这个方法返回 Matcher 本身，所以它可以用在方法链中（☞389）。

注 7：本书写作过程中，Sun 告诉我在“5.0u9”，也就是 Java 5.0 Update 9 中修正了此 bug（你可能还记得，365 页的脚注提到过本书针对的 Java 1.5 update 7）。在 Java 1.6 beta 中，这个 bug 也不存在。

Matcher **reset**(CharSequence *text*)

这个方法与 reset()差不多，但还会把目标文本改为新的 String（或者任何实现 CharSequence 的对象）。

如果你希望对多个文本应用同样的正则表达式（例如，对所读入的文件的每一行），使用 reset 方法比多次创建新的 Matcher 更有效率。

这个方法返回 Matcher 本身，所以可以用在方法链中（☞389）。

Pattern **pattern**()

Matcher 的 pattern 方法返回与此 Matcher 关联的 Pattern 对象。如果希望观察所使用的正则表达式，请使用 *m*.pattern().pattern()，它会调用 Pattern 对象（名字相同，但对象不同）的 pattern 方法（☞394）。

Matcher **usePattern**(Pattern *p*)

从 Java 1.5 开始添加，这个方法会用给定的 Pattern 对象替换当前与 Matcher 关联的 Pattern 对象。这个方法不会重置 Matcher，所以能够在文本的"当前位置"开始使用不同的 pattern。第 399 页有此方法实际应用的例子和讨论。

这个方法返回 Matcher 本身，所以可以用在方法链中（☞389）。

String **toString**()

从 Java 1.5 中添加，这个方法返回包含 Matcher 基本信息的字符串，调试时这很有用。字符串的内容可能会变化，在 Java 1.6 beta 中是这样：

```
Matcher m = Pattern.compile("(\\w+)").matcher("ABC 123");
System.out.println(m.toString());
m.find();
System.out.println(m.toString());
```

结果是：

```
java.util.regex.Matcher[pattern=(\w+) region=0,7 lastmatch=]
java.util.regex.Matcher[pattern=(\w+) region=0,7 lastmatch=ABC]
```

Java 1.4.2 的 Matcher 类只有继承自 java.lang.Object 的 toString 方法，它返回没什么信息含量的字符串：'java.util.regex.Matcher@480457'。

查询Matcher的目标字符串

Matcher类没有提供查询当前目标字符串的方法，但有些办法绕过了这种限制：

```
//此pattern在下面的函数中使用，此处编译并保存，是为了提高效率
static final Pattern pNeverFail = Pattern.compile("^");

//返回与matcher对象关联的目标字符串
public static String text(Matcher m)
{
   //记住这些位置，以备之后恢复
   Integer regionStart = m.regionStart();
   Integer regionEnd = m.regionEnd();
   Pattern pattern = m.pattern();

   //只有这样才能返回字符串
   String text = m.usePattern(pNeverFail).replaceFirst("");

   //恢复之前记录的位置
   m.usePattern(pattern).region(regionStart, regionEnd);

   // 返回文本
   return text;
}
```

这里使用`replaceFirst`方法，以及虚构的pattern和replacement字符串，来取得目标字符串的未经修改的副本。其中它重置了Matcher，但也恢复了之前的检索范围。它不是特别好的解决方案（效率也不是很高，而且即便Matcher的目标字符串可能是其他类型也会返回`String`），但是在Sun给出更好的办法之前，它还凑合。

Pattern的其他方法

Other Pattern Methods

除了主要的`compile` factories，`Pattern`类还提供了一些辅助方法：

`split`

下一页会详细讲解两种形式的`split`方法。

`String` **`pattern()`**

这个方法返回用于创建本`pattern`的正则表达式字符串。

`String` **`toString()`**

这个方法从Java 1.5之后可用，等价于`pattern`方法。

`int` **`flags()`**

这个方法返回`pattern`创建时传递给`compile` factory的flag参数（作为整数）。

`static String quote(String text)`

从 Java 1.5 开始提供的 static 方法，返回 text 对应的正则文字字符串，能够作为 `Pattern.compile` 的参数。例如，`Pattern.quote("main()")`返回字符串 ‘`\Qmain()\E`’，作为正则表达式它解释为「`\Qmain()\E`」，完全能够匹配原始的参数 ‘`main()`’。

`static boolean matches(String regex, CharSequence text)`

这个 static 方法返回的 Boolean 值表示正则表达式能否匹配文本(与 `matcher` 方法的参数一样，可以是一个 `String` 或者任何实现 `CharSequence` 的对象☛373)。其实，它等价于：

```
Pattern.compile(regex).matcher(text).matches();
```

如果需要传递编译参数，或者所要的信息不只是简单的匹配是否成功，则应该使用之前介绍的办法。

如果这个方法需要多次调用（例如，在循环中，或者其他经常调用的代码中），预先把正则表达式编译为一个 `Pattern` 对象，然后每次应用的效率会高很多。

Pattern 的 split 方法，单个参数

Pattern ' s split Method, with One Argument

`String[] split(CharSequence text)`

pattern 的这个方法接收**文本**（一个 `CharSequence` 对象），然后返回一个数组，包含由这个 pattern 对应的正则表达式在其中的匹配分隔的**文本**。`String` 类的 `split` 方法也提供了同样的功能。

```
String[] result = Pattern.compile("\\.").split("209.204.146.22");
```

返回 4 个字符串构成的数组（‘209’、‘204’、‘146’ 和 ‘22’），由文本中的三个「`\.`」分隔。这个简单例子只用单个字符分隔，但其实我们可以使用任何正则表达式。例如，你可以用非字母和数字的字符把一个字符串粗略地分为“单词”：

```
String[] result = Pattern.compile("\\W+").split(Text);
```

如果给出的字符串是“`what␣'␣s␣up,␣Doc`”，则返回 4 个字符串（‘`what`’、‘`s`’、‘`up`’、‘`Doc`’），由这个表达式的 3 次匹配（如果包含非 ASCII 文本，你可能需要使用「`\P{L}+`」或「`[^\p{L}\p{N}_]`」而不是「`\W+`」☛367）分隔。

相邻匹配之间的空元素

如果正则表达式能够在文本的最开头匹配，返回数组的第一个元素应该是空字符串（这是一个合法的字符串，但是不包括任何字符）。同样，如果正则表达式能够在某个位置匹配两次以上，应该返回空字符串，因为邻近的匹配"分割"了零长度的文本。例如：

```
String[] result = Pattern.compile("\\s*,\\s*").split(", one, two , ,, 3");
```

按照两边可能出现空白字符的逗号来分割，返回一个 5 个元素的数组：空字符串、'one'、'two'、两个空字符串和'3'。

序列末尾出现的所有空字符串都会被忽略：

```
String[] result = Pattern.compile(":").split(":xx:");
```

这样会得到两个字符串：一个空字符串和'xx'。如果希望保留这些元素，请使用下面介绍的双参数版本的 split。

Pattern 的 split 方法，两个参数

Pattern's split Method, with Two Arguments

```
String[] split(CharSequence text, int limit)
```

这个版本的 split 方法能够控制 pattern 应用的次数，以及结尾部分可能出现的空元素的处理策略。String 类的 split 方法也可以获得同样效果。

limit 参数等于 0，大于 0，还是小于 0，各有意义。

limit 小于 0

如果 *limit* 小于 0，就会保留数组结尾的空元素。

```
String[] result = Pattern.compile(":").split(":xx:", -1);
```

返回包含 3 个字符串的数组（一个空字符串，'xx'，然后是另一个空字符串）。

limit 等于 0

把 *limit* 设置为 0，等于不设置 *limit*，所以不会保留结尾的空元素。

limit 大于 0

如果 *limit* 大于 0，则 split 返回的数组最多包括 *limit* 个元素。也就是说，正则表达式至多会应用 *limit*-1 次。（如果 *limit*=3，则需要 2 次匹配来分割 3 个字符串）。

进行 *limit-1* 次匹配之后，匹配停止，目标字符串中其余的部分（在最后一次匹配之后的文本）会作为结果数组的尾元素。

如果某个字符串如下：

```
Friedl,Jeffrey,Eric Francis,America,Ohio,Rootstown
```

并且希望得到头三个部分，你可以将字符串分割为 4 个部分（头 3 个部分是名字，然后是最后的"其他"字符串）：

```
String[] NameInfo = Pattern.compile(",").split(Text, 4);
// NameInfo[0]是姓
// NameInfo[1]是名
// NameInfo[2]是中名(这里中名包括两个词)
// NameInfo[3]是其他部分，因为这里没用，直接忽略
```

为 `split` 设置 *limit* 的理由在于，如果不需要更多地处理，它可以用来停止查找其他的字符串、创建新字符串，限制数组的长度，提高效率。提供 *limit* 能够限制工作量。

拓展示例

Additional Examples

为 Image Tag 添加宽度和高度属性

Adding Width and Height Attributes to Image Tags

这里给出个高级的例子，原地（in-place）查找替换，修改 HTML，保证所有的 image tag 都包含 WIDTH 和 HEIGHT 属性（HTML 必须是 `StringBuilder`、`StringBuffer`，或者其他 `CharSequence`）。

只要有一副图像没有规定宽度或者高度，就可能降低整个页面的装载速度，因为浏览器在显示这幅图像之前必须读入整个文件。如果包含了宽度和高度的尺寸，文本和其他元素就可以立刻正确摆放，这样用户会感觉页面读取速度更快（注 8）。

如果找到 image tag，程序会寻找 SRC、WIDTH 和 HEIGHT 属性，如果存在，提取他们的值。如果 WIDTH 和 HEIGHT 有一个不存在，就先取回图像，计算尺寸，然后补充上属性。

如果 WIDTH 和 HEIGHT 都没有，就按照图像的真实尺寸来设置这些属性。如果存在一个，就只需要算出另一个属性，它的值是按比例计算出来的（例如，如果 WIDTH 是真实尺寸的一半，则添加的 HEIGHT 的值也是真实高度的一半，现代的浏览器就是这样处理的）。

注 8："所有的图像都必须有 size 属性"，这是 Yahoo!的规定，即使在早期也是如此。显然，今天仍然有许多流量巨大的站点页面的<img> tag 不包含尺寸信息。

和第 383 页的代码一样，这个程序手动维护匹配指针。它还使用了检索范围（☞384）和方法链（☞389）。代码如下：

```
// 匹配独立的<img> tags
Matcher mImg = Pattern.compile("(?id)<IMG\\s+(.*?)/?>").matcher(html);
// 匹配独立的 tag 中的 SRC、 WIDTH 和 HEIGHT 属性(所用的表达式很简单)
Matcher mSrc = Pattern.compile("(?ix)\\bSRC   =(\\S+)").matcher(html);
Matcher mWidth = Pattern.compile("(?ix)\\bWIDTH =(\\S+)").matcher(html);
Matcher mHeight= Pattern.compile("(?ix)\\bHEIGHT=(\\S+)").matcher(html);
int imgMatchPointer = 0; // 第一次搜索从字符串起始位置开始
while (mImg.find(imgMatchPointer))
{
   imgMatchPointer  = mImg.end(); // 下一次查找从这里开始
   // 在刚刚找到的 tag 中查找各字段
   Boolean hasSrc = mSrc.region( mImg.start(1), mImg.end(1) ).find();
   Boolean hasHeight = mHeight.region( mImg.start(1), mImg.end(1) ).find();
   Boolean hasWidth  =  mWidth.region( mImg.start(1), mImg.end(1) ).find();
   // 如果提供了 SRC 属性，但是没提供 WIDTH 和/或 HEIGHT ...
   if ( hasSrc && (! hasWidth || ! hasHeight))
   {
      java.awt.image.BufferedImage i = //获取图像
         javax.imageio.ImageIO.read(new java.net.URL(mSrc.group(1)));
      String size; //存放未提供的 WIDTH 和/或 HEIGHT 属性
      if (hasWidth)
          //得到了 width，根据比例计算 height
          size = "height='"  + (int)(Integer.parseInt(mWidth.group(1)) *
                                 i.getHeight() / i.getWidth()) + "' ";
      else if (hasHeight)
         // 得到了 height，根据比例计算 width
         size = "width='"    + (int)(Integer.parseInt(mHeight.group(1)) *
                                 i.getWidth() / i.getHeight()) + "' ";
      else // 两个属性都没提供，按实际情况处理
         size = "width='" + i.getWidth() + "' " +
                "height='" + i.getHeight() + "' ";
      html.insert(mImg.start(1), size); // 原地修改 HTML
      imgMatchPointer += size.length(); // 更新匹配指针
   }
}
```

尽管这例子很有教育意义，但还是有少数地方没考虑到。因为这个例子的重点在于本地查找和替换，尽可能地简化了其他方面，对需要处理的 HTML 进行了理想的假设。例如，正则表达式不容许属性的等号两边出现空白，属性中不会出现引号（参考第 202 页的 Perl 正则表达式，可以得到有实际意义的，Java 版本的 tag 属性匹配的办法）。这个程序不能处理相对 URL，也不能处理格式错误的 URL，也不能处理获取图像的代码抛出的任何异常。

不过，这个例子仍然说明了几个重要的概念。

对于每个 Matcher，使用多个 Pattern 校验 HTML

Validating HTML with Multiple Patterns Per Matcher

我们也可以用 Java 来写校验简单 HTML 的程序（☞132）。这段代码通过 usePattern 方法实时更换 Matcher 的 pattern。这样能够处理多个以「\G」开头的 pattern，进行对应的操作。请参考第 132 页的内容了解此方法的更多细节。

```
Pattern pAtEnd    = Pattern.compile("\\G\\z");
Pattern pWord     = Pattern.compile("\\G\\w+");
Pattern pNonHtml  = Pattern.compile("\\G[^\\w<>&]+");
Pattern pImgTag   = Pattern.compile("\\G(?i)<img\\s+([^>]+)>");
Pattern pLink     = Pattern.compile("\\G(?i)<A\\s+([^>]+)>");
Pattern pLinkX    = Pattern.compile("\\G(?i)</A>");
Pattern pEntity   = Pattern.compile("\\G&(#\\d+;\\w+);");

Boolean needClose = false;
Matcher m = pAtEnd.matcher(html); // 每个 Pattern 对象都能生成 Matcher 对象
while (! m.usePattern(pAtEnd).find())
{
   if (m.usePattern(pWord).find()) {
      ... m.group()包含一个单词或数值，可以进行对应的检查
   } else if (m.usePattern(pImgTag).find()) {
      ... 包含 image tag，检查是否合适...
   } else if (! needClose && m.usePattern(pLink).find()) {
      ... 有超链接，验证...
      needClose = true;
   } else if (needClose && m.usePattern(pLinkX).find()) {
      System.out.println("/LINK [" + m.group() + "]");
      needClose = false;
   } else if (m.usePattern(pEntity).find()) {
      // 容许出现&gt;和&#123;之类的 entity
   } else if (m.usePattern(pNonHtml).find()) {
      // 容许出现其他非单词的非 HTML 代码
   } else {
      // 完全无法匹配，肯定出了错，在此处选取一段文字用于错误输出
      m.usePattern(Pattern.compile("\\G(?s).{1,12}")).find();
      System.out.println("Bad char before '" + m.group() + "'");
      System.exit(1);
   }
}
if (needClose) {
   System.out.println("Missing Final </A>");
   System.exit(1);
}
```

因为 java.util.regex 的 bug，非 HTML 的匹配尝试即使不成功，仍然会“占用”目标字符串中的一个字符，所以我把这段程序放在最后。这个 bug 仍然存在，只是表现为错误输出中缺少第一个字符。我已经把这个 bug 提交给 Sun。

此 bug 没有修正之前，该如何使用单个参数的 find 方法来解决此问题呢？◆ 请翻到下页查看答案。

在单个参数的find()中使用多个Pattern

◆ 第399页问题的答案

在第399页的程序中，java.util.regex错误地设置了matcher的“当前位置”，所以下一次查找会从错误的位置开始。我们可以绕过这个bug，自己记录“当前位置”，使用单个参数的find从此位置开始查找。

程序中修改的部分以高亮显示：

```
Pattern pWord       = Pattern.compile("\\G\\w+");
Pattern pNonHtml    = Pattern.compile("\\G[^\\w<>&]+");
Pattern pImgTa      = Pattern.compile("\\G(?i)<img\\s+([^>]+)>");
Pattern pLink       = Pattern.compile("\\G(?i)<A\\s+([^>]+)>");
Pattern pLinkX      = Pattern.compile("\\G(?i)</A>");
Pattern pEntity     = Pattern.compile("\\G&(#\\d+|\\w+);");
Boolean needClose   = false;
Matcher m = pWord.matcher(html); // 每个Pattern对象都能生成Matcher对象
Integer currentLoc = 0;
while (currentLoc < html.length())
{
  if (m.usePattern(pWord).find(currentLoc)) {
     ... m.group()包含一个单词或数值，可以进行对应的检查
  } else if (m.usePattern(pImgTag).find(currentLoc)) {
     ... 包含image tag，检查是否合适...
  } else if (! needClose && m.usePattern(pLink).find(currentLoc)) {
     ... 有超链接，验证...
     needClose = true;
  } else if (needClose && m.usePattern(pLinkX).find(currentLoc)) {
     System.out.println("/LINK [" + m.group() + "]");
     needClose = false;
  } else if (m.usePattern(pEntity).find(currentLoc)) {
     // 容许出现&gt;和&#123;之类的entity
  } else if (m.usePattern(pNonHtml).find(currentLoc)) {
     // 容许出现其他非单词的非HTML代码
  } else {
     // 完全无法匹配，肯定出了错，在此处选取一段文字用于错误输出
     m.usePattern(Pattern.compile("\\G(?s).{1,12}")).find(currentLoc);
     System.out.println("Bad char at '" + m.group() + "'");
     System.exit(1);
  }
  currentLoc = m.end(); // “当前位置”指向上次匹配的结尾
}
if (needClose) {
  System.out.println("Missing Final </A>");
  System.exit(1);
}
```

与之前的程序不同，这里调用find时指定了检索的开始偏移值，所以不必指定region。不过，你可以自己维护这个region，在每次find之前恰当地调用region。

```
m.usePattern(pWord).region(start, end).find(currentLoc)
```

解析 CSV 文档

Parsing Comma-Separated Values (CSV) Text

这里是用 java.util.regex 写的解析 CSV 的例子（参见第 6 章☞271）。这里的程序使用占有优先量词（☞142)取代固化分组，因为这样看起来更清楚。

```
String regex = // 双引号字段保存到 group(1)，非引号字段保存到 group(2)
    "  \\G(?:^|,)                                  \n"+
    "  (?:                                         \n"+
    "       # 要么是双引号字段 ...                  \n"+
    "       \" # 开头双引号                         \n"+
    "       ( [^\"]*+ (?: \"\" [^\"]*+ )*+ )        \n"+
    "       \" # 结束双引号                         \n"+
    "  |# ... 或者是 ...                            \n"+
    "        # 非引用，非逗号文本 ...               \n"+
    "        ([^\",]*+)                             \n"+
    "  )                                           \n";
// 根据上面的表达式创建 matcher，解析其中的一行 CSV 文档
Matcher mMain = Pattern.compile(regex, Pattern.COMMENTS).matcher(line);
// 创建匹配「""」的 matcher，目标文本暂时为虚构
Matcher mQuote = Pattern.compile("\"\"").matcher("");

while (mMain.find())
{
    String field;
    if (mMain.start(2) >= 0)
        field = mMain.group(2); // 非引号字段，直接使用
    else
       // 引用字段，用单引号替换两个相连的引号
       field = mQuote.reset(mMain.group(1)).replaceAll("\"");

    // 现在处理字段的内容 ...
    System.out.println("Field [" + field + "]");
}
```

这个程序比第 217 页的原始程序效率要高很多，原因在于：正则表达式效率更高，按照第 6 章第 271 页介绍的办法，重复使用单个 Matcher（通过单个参数版本的 reset 方法），而没有不断创建-回收 Mather。

Java 版本差异

Java Version Differences

本章开头已经提到，本书主要针对 Java 1.5.0。不过，目前 Java 1.4.2 仍然在广泛应用，而 Java 1.6 已经整装待发（已经发布了 beta 版，但不会很快发布正式版）。所以，我得简要地提一下 1.4.2 和 1.5.0（Update 7）之间的差异，以及 1.5.0 和目前的 1.6 "build 59g" 的差异。

1.4.2和1.5.0之间的差异

Differences Between 1.4.2 and 1.5.0

相对Java 1.4.2，Java 1.5.0添加了许多新的方法。大多数新的方法主要是为了支持Matcher的检索范围。此外，Unicode支持也升级并提高了效率。所有的变化都在下面两节详细介绍（注9）。

1.5.0中的新方法

与检索范围相关的Matcher方法都没有出现在Java 1.4.2中：

- `region`
- `regionStart`
- `regionEnd`
- `useAnchoringBounds`
- `hasAnchoringBounds`
- `useTransparentBounds`
- `hasTransparentBounds`

Java 1.4.2也不包含下面的方法：

- `toMatchResult`
- `hitEnd`
- `requireEnd`
- `usePattern`
- `toString`

Java 1.4.2不包含下面这个static方法：

- `Pattern.quote`

1.4.2和1.5.0关于Unicode支持的差异

1.4.2和1.5.0在Unicode相关的问题上有这些变化：

- Java 1.4.2的Unicode使用的是Unicode Version 3.0.0，1.5.0使用的是Unicode Version 4.0.0。这个改变影响到很多方面，例如字符的定义（例如，在Version 3.0.0中，`\uFFFF`之后是没有代码点的），以及属性和Unicode区块的定义。

- 增强了通过「`\p{…}`」和「`\P{…}`」引用区块的方式（参加Java关于`Character.UnicodeBlock`的文档，得到区块列表及其正式名称）。

注9：递归结构「`(?1)`」未在文档中说明，在Java 1.4.2中是非正式提供的，在1.5.5中已经不再包含。PCRE也是如此，但是Java没有在文档中说明，我们在这里用脚注来说明。

Java 1.4.2 中的规则是“去掉官方代码块名字中的空格，和开头的 In”。这样，区块引用就类似\p{InHangulJamo}和\p{InArabicPresentationForms-A}。

1.5.0 添加了两种新的区块命名。可以在官方块名称之前直接添加‘In’，所以名字可以为\p{InHangul·Jamo}和\p{InArabic·Presentation·Forms-A}。也可以给 Java 的区块标识符添加前缀‘In’（是官方名称，将空格和连字符替换为下画线）\p{InHangul_Jamo}和 \p{InArabic_Presentation_Forms_A}。

- Java 1.4.2 存在一个古怪的 bug，**Arabic Presentation Forms-B** 和 **Latin Extended-B** 结尾的“B”必须写作“Bound”，也就是说\p{InArabicPresentationForms-Bound}和\p{InLatinExtended-Bound}。
- Java 1.5.0 中 Character 类的 *isSomeing* 方法支持正则表达式（☞369）。

1.5.0 和 1.6 之间的差异

Differences Between 1.5.0 and 1.6

写作本书时已经发布的 1.6 和 1.5.0 在正则表达式的问题上只有两个细微的差别：

- Java 1.6 提供了之前没有的对 **Pi** 和 **Pf** 的 Unicode 分类的支持。
- 「\Q…\E」结构的 bug 已经修正了，所以在字符组中可以正常工作。





第 9 章 .NET

.NET

Microsoft 的.NET Framework 中可以使用 Visual Basic、C#和 C++(以及其他许多语言)，.NET 提供了公用的正则表达式库，统一了不同语言之间的正则表达式语意。它的引擎特性完备，功能强大，容许我们在速度和便利之间求得最大的均衡（注 1)。

每种语言在处理对象和方法时都有不同的语意，但是某些基本的对象和方法在所有语言中都是相通的，所以不管使用哪种语言编写的复杂例子，都可以直接转换到.NET 语言套件中的其他语言中。本章中的例子使用 Visual Basic。

与之前各章的联系 在开始本章的内容之前必须说明，第 1 到 6 章的基础知识对理解本章非常重要。我猜测，有些只对.NET 有兴趣的读者可能会从本章开始阅读这本书，我希望他们认真地读一读前言（尤其是体例部分）和前面的章节：第 1、2、3 章介绍了与正则表达式相关的基本概念、特性和技术，第 4、5、6 章介绍了一些理解正则表达式的关键知识，它们可以直接应用到.NET 的正则表达式中。前几章讲解的重要概念包括 NFA 引擎进行匹配的基本原理、匹配优先性、回溯和效率。

接下来要强调的是，除了用于速查列表——例如本章的第 407 页，和第 3 章从第 114 页到第 123 页，我并不希望这本书成为参考手册，而希望它成为精通正则表达式的详细教科书。



注 1：本书针对.NET Framework Version2.0（随同 Visual Studio 2005 一起发售)。

本章首先介绍.NET 的正则流派，包括元字符的支持事宜，以及.NET 程序员必须面对的特殊问题。然后是总括.NET 中正则表达式相关的对象模型，详细讲解居于核心地位的，与正则表达式相关的类。最后用例子来说明，如何将预先构建好的正则表达式封装到共享的装配件（assembly）中，组成个人的正则表达式库。

.NET 的正则流派

.NET's Regex Flavor

.NET 使用的是传统型 NFA 引擎，所以第 4、5、6 章讲解的 NFA 的知识都适用于.NET。下一页的表 9-1 简要说明了.NET 的正则流派，其中大部分已经在第 3 章介绍过。

在接收正则表达式的函数和结构中设置标志位（flag），或是在正则表达式之内使用「(?modes-modes)」和「(?mods-mods:…)」结构，可以使用不同的**匹配模式**，流派的许多方面也会因此发生变化（☛110）。408 页的表 9-2 列出了这些模式。

其中包括了「\w」之类的"纯"转义。它们可以直接用到 VB.NET 的字符串文字（"\w"）和 C#的 verbatim 字符串（@"\w"）中。C++的语言没有提供针对正则表达式的字符串文字，所以正则表达式中的反斜线在字符串文本中需要双写（"\\w"）。请参考"作为正则表达式的字符串"（☛101）。

下面是对表 9-1 的补充说明：

① \b 只有在字符组内部才作为退格符。在字符组之外，\b 匹配单词分界符（☛133）。

\x##容许出现两位十六进制数字，例如「**\xFC**ber」匹配'über'。

\u####容许且只容许四位十六进制数字，例如「**\u00FC**ber」匹配'über'，「\u20AC」匹配'€'。

② .NET Framework Version 2.0 中的字符组支持**集合减法**，例如「[a-z-[aeiou]]」表示小写的非元音 ASCII 字母（☛125）。在字符组内部，连字符之后又跟着字符组表示字符组的减法运算，减去后面字符组内部的字符。

③ **\w**、**\d** 和**\s**（以及对应的**\W**、**\D** 和**\S**）通常能处理所有合适的 Unicode 字符，但是如果启用了 RegexOptions.ECMAScript（☛412），就只能处理 ASCII 字符。

在此默认模式下，**\w** 匹配 Unicode 属性\p{Ll}、\p{Lu}、\p{Lt}、\p{Lo}、\p{Nd}和\p{Pc}。请注意，其中并没有\p{Lm}（请参考第 123 页的属性列表）。

表 9-1：NET 正则表达式流派概览

<table>
<tr><th colspan="2">字符缩略表示法①</th></tr>
<tr><td>☞115 (c)</td><td>\a [\b] \e \f \n \r \t \v\octal \x## \u#### \cchar</td></tr>
<tr><th colspan="2">字符组及相关结构</th></tr>
<tr><td>☞118</td><td>字符组②：[…] [^…]（可包含集合运算符☞125）</td></tr>
<tr><td>☞119</td><td>几乎任何字符：点号（根据模式的不同，有各种含义）</td></tr>
<tr><td>☞120 (c)</td><td>字符组缩略表示法③：\w \d \s \W \D \S</td></tr>
<tr><td>☞121 (c)</td><td>Unicode 属性和区块④：\p{Prop} \P{Prop}</td></tr>
<tr><th colspan="2">锚点及其他零长度断言</th></tr>
<tr><td>☞129</td><td>行/字符串起始位置：^ \A</td></tr>
<tr><td>☞129</td><td>行/字符串结束位置：$ \z \Z</td></tr>
<tr><td>☞130</td><td>当前匹配的起始位置⑤：\G</td></tr>
<tr><td>☞133</td><td>单词分界符：\b \B</td></tr>
<tr><td>☞133</td><td>环视结构⑥：(?=…) (?!…) (?<=…) (?<!…)</td></tr>
<tr><th colspan="2">注释及模式修饰符</th></tr>
<tr><td>☞135</td><td>模式修饰符：(?mods-mods) 容许出现的模式：x s m i n (☞408)</td></tr>
<tr><td>☞135</td><td>模式修饰范围：(?mods-mods:…)</td></tr>
<tr><td>☞136</td><td>注释：(?#…)</td></tr>
<tr><th colspan="2">分组及捕获</th></tr>
<tr><td>☞137</td><td>捕获型括号⑦：(…) \1 \2…</td></tr>
<tr><td>☞436</td><td>对称分组：(?<<i>name-name</i>>…)</td></tr>
<tr><td>☞409</td><td>命名捕获及回溯：(?<<i>name</i>>…) \k<<i>name</i>></td></tr>
<tr><td>☞137</td><td>仅分组的括号：(?:…)</td></tr>
<tr><td>☞139</td><td>固化分组：(?>…)</td></tr>
<tr><td>☞139</td><td>多选结构：|</td></tr>
<tr><td>☞141</td><td>匹配优先量词：* + ? {n} {n,} {x,y}</td></tr>
<tr><td>☞141</td><td>忽略优先量词：*? +? ?? {n}? {n,}? {x,y}?</td></tr>
<tr><td>☞109</td><td>条件判断：(?if then|else)——"if"部分可以是环视、(num)或(name)</td></tr>
<tr><td>(c) ——可用于字符组内部</td><td>①……⑦见说明</td></tr>
</table>

在默认模式下，\s 匹配「[·\f\n\r\t\v\x85\p{Z}]」，U+0085 是 Unicode 中的 NEXT LINE 控制字符，\p{Z} 匹配 Unicode 的"分隔符"字符（☞122）。

④ \p{…}和\P{…}支持标准的 Unicode 属性和区块（针对 Unicode Version 4.0.1）。不支持 Unicode 字母表。

区块名要求出现'Is'前缀（参考第 125 页的表格），只能够使用含有空格或者下画线的格式。例如，\p{Is_Greek_Extended}和\p{Is Greek Extended}是不容许的，正确的只有\p{IsGreekExtended}。

.NET 只支持\p{Lu}之类的短名称，而不支持\p{Lowercase_Letter}之类的长名称。单字母属性也要求使用花括号（也就是说，不能把\p{L}简记为\pL）。请参考第 122 和第 123 页的表格。

.NET 也不支持特殊的复合属性\p{L&}，以及特殊属性\p{All}、\p{Assigned}和\p{Unassigned}。相反，你可以使用「(?s:.)」、「\P{Cn}」、「\p{Cn}」分别来代替。

⑤ \G 表示上一次匹配的**结束**位置，虽然文档介绍说它表示**本次**匹配的**开头**位置（☞130）。

⑥ 顺序环视和逆序环视中都可以使用任意形式的正则表达式。就我所知，.NET 正则引擎是唯一容许在逆序环视中出现能够匹配任意长度文本表达式的引擎（☞133）。

⑦ RegexOptions.ExplicitCapture 选项（也可通过模式修饰符(?n)设定）会禁止普通的「(…)」括号的捕获功能。不过明确命名的捕获型括号——例如「(?<num>\d+)」——仍然有效（☞138）。如果使用了命名分组，此选项容许你使用更加美观的「(…)」，而不是「(?:…)」，来进行纯粹的分组。

表 9-2：NET 的匹配模式和正则表达式模式

RegexOptions 选项	(?mode)	说明
.Singleline	s	点号能匹配任何字符（☞111）
.Multiline	m	扩展「^」和「$」的匹配（☞111）
.IgnorePatternWhitespace	x	设置宽松排列和注释模式（☞72）
.IgnoreCase	i	进行不区分大小写的匹配
.ExplicitCapture	n	关闭「(…)」的捕获功能，只有「(?<*name*>…)」能够捕获（☞412）
.ECMAScript		限制「\w」、「\s」和「\d」只对 ASCII 字符有效（☞412）
.RightToLeft		传动装置的驱动过程不变，但是方向相反（从字符串的末尾开始，向开头移动）。不幸的是这个选项会有问题（☞411）
.Compiled		多花些时间优化正则表达式，这样应用时匹配更迅速（☞410）

对于流派的补充

Additional Comments on the Flavor

下面介绍一些其他的相关细节。

命名捕获

.NET 支持命名捕获（☞138），它通过「(?<*name*>…)」或是「(?'*name*'…)」实现。这两种办法是等价的，可以随意选用其中一种，不过我更喜欢<…>，因为我相信使用它的人多一些。

要反向引用命名捕获匹配的文本，可以使用「\k<*name*>」或是「\k'*name*'」。

在匹配之后（也就是是 `Match` 对象生成之后；下文从第 416 页开始概要介绍.NET 的对象模型），命名捕获匹配的文本可以通过 `Match` 对象的 `Groups(`*name*`)` 属性来访问（C#使用 `Groups[`*name*`]`）。

在 replacement 字符串中（☞424），命名捕获的结果通过 `${`*name*`}` 来访问。

某些情况下，可能需要按数字顺序访问所有的分组，所以命名捕获的分组也会被标上序号。它们的编号从所有未命名的分组之后开始：

```
 1  1 3          3 2   2
「(\w) (?<Num>\d+)(\s+)」
```

本例中，我们可以用 `Groups("Num")` 或 `Groups(3)` 来访问「\d+」匹配的文本。这两个名字对应同一个分组。

不幸的结果

一般情况下不应该把正常的捕获型括号和命名捕获混合起来，不过如果你这样做了，就必须彻底理解捕获分组的编号顺序。如果捕获型括号用于 `Split`（☞425），或者在 replacement 字符串中使用了‘`$+`’（☞424），编号就很重要。

条件测试

如果「(?*if then*|*else*)」中的 *if* 部分（☞140）可以使用任意类型的环视结构，也可以在括号中使用捕获分组的编号，或者是命名分组的名字。这里出现的纯文本（或者纯正则表达式）会被自动当作肯定型顺序环视来处理（也就是说，可以将其看作「(?=…)」包围的结构）。这可能带来麻烦：例如，「…(?(Num) *then*|*else*)…」中的「(Num)」会变为「(?=Num)」（也就是顺序环视的‘`Num`’），如果在正则表达式的其他地方没有出现「(?<Num>…)」时会这样。如果存在这样的命名捕获，*if* 判断的就是它是否捕获成功。

我推荐不要依赖这种“自动化顺序环视（auto-lookaheadfication）”，而明确使用「(?=…)」把意图传达给看程序的人，这样如果正则引擎在未来修改了 *if* 语法，也不会带来意外。

“编译好的”正则表达式

在前面几章，我使用“编译（compile）”这个词来描述所有正则表达式系统中，在应用正则表达式之前必须做的准备工作，它们用来检查正则表达式是否格式规范，并将其转换为能够实际应用的内部形式。在.NET 的正则表达式中，它的术语是“解析（parsing）”。.NET 使用两种意义的“编译”来指涉解析阶段的优化。

下面是增进优化效果的细节：

- **解析**（Parsing） 程序在执行过程中，第一次遇到正则表达式时必须检查它是否格式规范，并将其转换为适于正则引擎实际应用的内部形式。此过程在本书的其他部分称为“编译（compile）”。

- **即用即编译**（On-the-Fly Compilation） 在构建正则表达式时，可以指定 `RegexOptions.Compiled` 选项。它告诉正则引擎，要做的不仅是此表达式转换为某种默认的内部形式，而是**编译**为底层的 MSIL（Microsoft Intermediate Language）代码，在正则表达式实际应用时，可以由 JIT（“Just-In-Time”编译器）优化为更快的本地机器代码。

  这样做需要花费更多的时间和空间，但这样得到的正则表达式速度更快。本节之后会讨论这样的权衡。

- **预编译的正则表达式** 一个（或多个）Regex 对象能够封装到 DLL（Dynamically Loaded Library，例如共享的库文件）中，保存在磁盘上。这样其他的程序也可以直接调用它。称为“编译装配件（assembly）”。请参考“正则表达式装配件”（☞434）获得更多信息。

如果使用 `RegexOptions.Compiled` 来进行“即用即编译”的编译，在启动速度，持续内存占用和匹配速度之间，存在此消彼长的关系：

标　准	不使用 RegexOptions.Compiled	使用 RegexOptions.Compiled
启动速度	较快	较慢（最多提升 60 倍）
内存占用	少	多（每个表达式占用 5-15KB）
匹配速度	一般	最多能提升 10 倍

在程序第一次遇到正则表达式时进行初始的正则表达式解析（默认情况，即不用 `RegexOptions.Compiled`）相对来说是很快的。即使在我这台有年头的 550MHz NT 的机器上，每秒钟也能进行大约 1 500 次复杂编译。如果使用 `RegexOptions.Compiled`，则速度下降到每秒 25 次，每个正则表达式需要多占用大约 10KB 内存。

更重要的是，在程序的执行过程中，这块内存会一直占用——它无法释放。

在对时间要求不严格的场合使用 RegexOptions.Compiled 无疑是很有意义的，在这里，速度是很重要的，尤其是需要处理大量的文本时更是如此。另一方面，如果正则表达式很简单，需要处理的文本也不是很多，这样做就没有意义。如果情况不是这样黑白分明，该如何选择就不那么容易了——必须具体情况具体分析，以进行取舍。

某些情况下，把编译的正则表达式作为“编译好的”正则对象封装到 DLL 中是很有价值的。最终的程序所占的内存更少（因为不必装载编译正则表达式所需的包），装载速度更快（因为在 DLL 生成时它们已经编译好了，只需要直接使用即可）。另一个不错的副产品就是，表达式还可以供其他需要的程序使用，所以这是一种组建个人正则表达式库的好办法。请参考第 435 页的“使用装配件构建自己的正则表达式库”。

从右向左的匹配

长期以来，正则表达式的开发人员一直觊觎着“反向（backwards）”匹配（即从右向左，而不是从左向右）。对开发人员来说，最大的问题可能是，“从右向左”的匹配到底是什么意思？是整个正则表达式都需要反过来吗？还是说，这个正则表达式仍然在目标字符串中进行尝试，只是传动装置从结尾开始，驱动过程从右向左进行？

抛开这些纯粹的概念，看个具体的例子：用「\d+」匹配字符串‘123•and•456’。我们知道正常情况下结果是‘123’，根据直觉，从右向左匹配的结果应该是‘456’。不过，如果正则引擎使用的规则是，从字符串末尾开始，驱动过程从左向右进行，结果可能就会出乎意料。在某些语意下，正则引擎能够正常工作（从开始的位置向右“看”），所以第一次尝试「\d+」是在‘…456▲’，这里无法匹配。第二次尝试在‘…45▲6’，这里能够匹配，所以驱动过程开始“考察”位置‘6’，这当然可以匹配「\d+」，所以最后的结果是‘6’。

.NET 的正则表达式提供了 RegexOptions.RightToLeft 的选项。但它究竟是什么意义呢？答案是：“这问题值得思索。”它的语意没有文档，我测试了也无法找到规律。在许多情况下——例如‘123•and•456’，它给出符合直觉的结果（也就是‘456’）。

不过，有时候也会报告没有匹配结果，或是匹配跟其他结果相比毫无意义的文本。

如果需要进行从右向左的匹配，你可能会发现，RegexOptions.RightToLeft 似乎能得到你期望的结果，但是最后，你会发现这样做得冒风险。

反斜线-数字的二义性

数字跟在反斜线之后，可能表示十进制数的转义，也可能是反向引用。到底应该如何处理，取决于是否指定了 RegexOptions.ECMAScript 选项。如果你不关心其中的细微差别，不妨一直用「\k<*num*>」表示反向引用，或者在表示十进制数时以 0 开头（例如「\08」）。这两种办法不受 RegexOptions.ECMAScript 的影响。

如果没有使用 RegexOptions.ECMAScript，从「\1」到「\9」的单个转义数字通常代表反向引用，而以 0 开头的转义数字通常代表十进制转义（例如，「\012」匹配 ASCII 的进纸符 linefeed)，除此之外的所有情况下，如果"有意义"（也就是说某个正则表达式中有足够多的捕获型括号），数字都会被作为反向引用来处理。否则，如果数字的值处于\000 和\377 之间，就作为十进制转义。例如，如果捕获型括号的数目多于 12，则「\12」会作为反向引用，否则就会作为十进制数字。

下一节详细讲解 RegexOptions.ECMAScript 的语意。

ECMAScript 模式

ECMAScript 的基础是一种标准版本的 JavaScript（注 2），还包含了它自己的解析和应用正则表达式的语意。如果使用 RegexOptions.ECMAScript 选项，.NET 的正则表达式就会模拟这些语意。如果你不明白 ECMAScript 的含义，或者不需要兼容它，就完全可以忽略该节。

如果启用了 RegexOptions.ECMAScript，将会应用下面的规则：

- RegexOptions.ECMAScript 只能与下面的选项同时使用：

```
RegexOptions.IgnoreCase
RegexOptions.Multiline
RegexOptions.Compiled
```

- \w、\d、\s、\W、\D、\S 只能匹配 ASCII 字符。

注 2：ECMA 表示"European Computer Manufactures Association（欧洲计算机制造商协会）"，成立于 1960 年，任务是为不断发展计算机的各个方面制定标准。

- 正则表达式中的反斜线-数字的序列不会有反向引用和十进制转义的二义性，它只能表示反向引用，即使这样需要截断结尾的数字。例如，「(…)\10」中的「\10」会被处理为，第 1 组捕获性括号匹配的文本，然后是文字‘0’。

使用.NET 正则表达式

Using .NET Regular Expressions

.NET 正则表达式功能强大，语法清晰，通过完整而易于使用的类接口来操作。虽然微软的正则表达式包做得很漂亮，文档却相反——它非常糟糕。文档不够全面，编写不够清晰，缺乏组织，有时甚至不能保证正确性。我花了很长的时间才整理清楚，所以希望这一章的内容能够让读者更清楚地理解.NET 的正则表达式。

正则表达式快速入门

Regex Quickstart

即使不需要知道正则类模型（regex class model）的细节，也可以直接上手使用.NET 的正则表达式包。理解细节能够让我们获得更多的信息，提高工作效率，但是下面这些简单的例子没有明确创建任何正则类，细节将在例子之后提到。

使用正则表达式库的程序必须在文件的开头写上下面这条语句，下面的例子假设此句已经存在：

```
Imports System.Text.RegularExpressions
```

下面的例子都能正常处理 `String` 变量 `TestStr`。本章的所有例子中，选用的变量名都以斜体标注。

快速入门：在字符串中寻找匹配

这段程序检查一个正则表达式是否能匹配字符串：

```
If Regex.IsMatch(TestStr, "^\s*$")
   Console.WriteLine("line is empty")
Else
   Console.WriteLine("line is not empty")
End If
```

这个例子使用了匹配模式：

```
If Regex.IsMatch(TestStr, "^subject:", RegexOptions.IgnoreCase)
   Console.WriteLine("line is a subject line")
Else
   Console.WriteLine("line is not a subject line")
End If
```

快速入门：匹配，获得匹配文本

这个例子显示正则表达式实际匹配的文本。如果没有匹配，TheNum 就是空字符串。

```
Dim TheNum as String = Regex.Match(TestStr, "\d+").Value
If TheNum <> ""
    Console.WriteLine("Number is: " & TheNum)
End If
```

这个例子使用了一个匹配模式：

```
Dim ImgTag as String = Regex.Match(TestStr, "<img\b[^>]*>", _
                                   RegexOptions.IgnoreCase).Value
If ImgTag <> ""
   Console.WriteLine("Image tag: " & ImgTag)
End If
```

快速入门：匹配，获得捕获文本

这段程序以字符串的形式返回第 1 个捕获分组的匹配文本：

```
Dim Subject as String = _
    Regex.Match(TestStr, "^Subject: (.*)").Groups(1).Value
If Subject <> ""
   Console.WriteLine("Subject is: " & Subject)
End If
```

请注意，在 C#中应使用 Groups[1]取代 Groups(1)。

下面的程序目的相同，只是使用了 match 选项：

```
Dim Subject as String = _
    Regex.Match(TestStr, "^subject: (.*)", _
                RegexOptions.IgnoreCase).Groups(1).Value
If Subject <>  ""
    Console.WriteLine("Subject is: " & Subject)
End If
```

仍然是相同的程序，只是使用命名捕获：

```
Dim Subject as String = _
    Regex.Match(TestStr, "^subject: (?<Subj>.*)", _
                 RegexOptions.IgnoreCase).Groups("Subj").Value
If Subject <> ""
   Console.WriteLine("Subject is: " & Subject)
End If
```

快速入门：查找和替换

这个例子把输入的字符串转换为 HTML“安全”的字符，把特殊的 HTML 转换为 HTML entity：

```
TestStr = Regex.Replace(TestStr, "&", "&amp;")
TestStr = Regex.Replace(TestStr, "<", "&lt;")
TestStr = Regex.Replace(TestStr, ">", "&gt;")
Console.WriteLine("Now safe in HTML: " & TestStr)
```

replacement 字符串（第 3 个参数）的处理是很特殊的，第 424 页的补充内容做了讲解。例如，在 replacement 字符串中，‘$&’会被正则表达式真正匹配的文本所替代，下面的例子给大写的单词添加<B>…</B>：

```
TestStr = Regex.Replace(TestStr, "\b[A-Z]\w*", "<B>$&</B>")
Console.WriteLine("Modified string: " & TestStr)
```

这个例子把<B>…</B>（使用不区分大小写的匹配）替换为<I>…</I>：

```
TestStr = Regex.Replace(TestStr, "<b>(.*?)</b>", "<I>$1</I>", _
                        RegexOptions.IgnoreCase)
Console.WriteLine("Modified string: " & TestStr)
```

包概览

Package Overview

通过丰富而便捷的类结构，可以使用.NET 正则表达式几乎所有的功能。下面这个完整的控制台程序，提供了关于整个包的概览，它明确使用各种对象来进行简单的匹配。

```
Option Explicit On ' 使用正则表达式时并非必须这样写
Option Strict On   ' 但这样做是个好习惯
' 简化正则表达式相关的类的访问
Imports System.Text.RegularExpressions

Module SimpleTest
Sub Main()
    Dim SampleText as String = "this is the 1st test string"
    Dim R as Regex = New Regex("\d+\w+") '编译这个pattern
    Dim M as Match = R.match(SampleText) '应用到字符串中
    If not M.Success
        Console.WriteLine("no match")
    Else
        Dim MatchedText as String  = M.Value '查询结果...
        Dim MatchedFrom as Integer = M.Index
        Dim MatchedLen  as Integer = M.Length
        Console.WriteLine("matched [" & MatchedText & "]" & _
                          " from char#" & MatchedFrom.ToString() & _
                          " for " & MatchedLen.ToString() & " chars.")
    End If
  End Sub
  End Module
```

通过命令行提示符来执行，把「\d+\w+」应用到同样的文本，结果是：

```
matched [1st] from char#12 for 3 chars.
```

导入正则表达式名字空间

你注意到程序头部的 `Imports System.Text.RegularExpressions` 了吗？任何用到.NET 正则对象的 VB 程序都必须写上这一条语句，才能通过编译。

C#中对应的是：

```
using System.Text.RegularExpressions; //C#中应这么写
```

这个例子说明了基本的正则对象的用法，下面两行主要行为：

```
Dim R as Regex = New Regex("\d+\w+") '编译这个pattern
Dim M as Match = R.match(SampleText) '应用到字符串中
```

也可以合并为一行：

```
Dim M as Match = Regex.Match(SampleText, "\d+\w+") '对字符串应用pattern
```

合并的写法更容易使用，程序员需要输入的代码更少，需要记录的对象也更少。不过，它的效率会稍微低一些（☞432）。在下面几页中，我们首先会看到原始的对象，然后学习“便捷”函数，例如静态函数 Regex.Match，以及合适的使用时机。

为简便起见，在程序片段的例子中，我会省略下面这几行：

```
Option Explicit On
Option Strict On
Imports System.Text.RegularExpressions
```

本书的第96、99、204、219和237页出现过VB的例子，回顾它们也许有所帮助。

核心对象概览
Core Object Overview

在深入细节之前，先来看看.NET的正则对象模型。对象模型是一套类结构，正则表达式的各种功能通过它们来提供。.NET的正则功能通过7个高度交互的类来提供，实际上你可能只需要理解其中3个——也就是下一页的图9-1所示的3个类——即可，它们展示了对‘May·16,·1998’重复应用「\s+(\d+)」的过程。

Regex对象

第一步是创建 Regex 对象，例如：

```
Dim R as Regex = New Regex("\s+(\d+)")
```

在这里，我们用一个 Regex 对象表示「\s+(\d+)」，将其保存在变量R中。获得 Regex 对象之后，我们可以通过 Match(*text*) 方法将其应用到一段文本，返回与第一次匹配结果相关的信息：

```
Dim M as Match = R.Match("May 16, 1998")
```

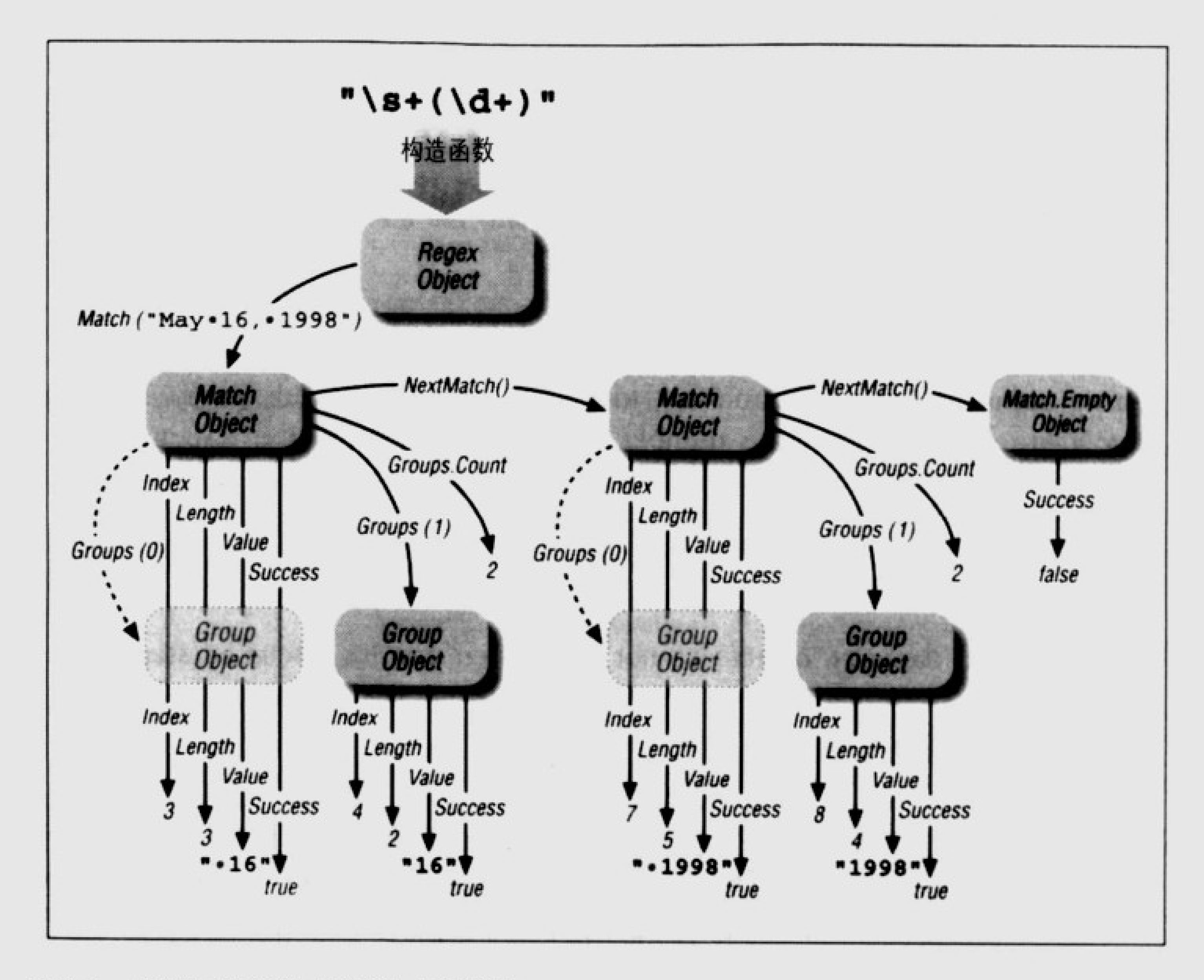


图 9-1：.NET 正则表达式相关对象模型

Match 对象

Regex 对象的 Match(…) 方法通过创建并返回 Match 对象来提供匹配信息。Match 对象有多个属性，包括 Success（一个表示匹配是否成功的 Boolean 值）和 Value（如果匹配成功，则保存实际匹配文本的副本）。稍后我们会看到 Match 的所有属性的列表。

Match 对象返回的细节中还包括捕获型括号所匹配的文本。前面 Perl 的例子使用$1 保存第一组捕获型括号匹配的文本。.NET 提供了两种办法：如果要取得纯文本，可以按照索引值访问 Match 对象的 Groups 属性，例如 Groups(1).Value，它等价于 Perl 的$1（请注意，C#中使用的是 Groups[1].Value）。另一个办法是使用 Result 方法，请参考第 429 页。

Group 对象

前一段中的`Groups(1)`其实是对`Group`对象的引用，后面的`.Value`引用它的`Value`属性（也就是此分组对应的文本）。每一组捕获型括号都对应一个`Group`对象，另外还有一个"虚拟分组（virtual group）"，其编号为0，它保存全局匹配的信息。

因此，*MatchObj.Value* 和 *MatchObj*`.Groups(0).Value`是等价的——都是全局匹配的文本的副本。第一种写法更加简洁方便，但我们必须知道存在编号为0的分组，因为 *MatchObj*`.Groups.Count`（也就是`Match`关联的分组的数目）包含了它。如果「`\s+(\d+)`」能够匹配成功，*MatchObj*`.Groups.Count`的值就是2（标号为0的全局匹配和`$1`）。

Capture 对象

`Capture`对象的使用并不频繁，请参考第437页的介绍。

匹配时会计算出所有结果

把正则表达式应用到字符串中，得到一个`Match`对象，此时所有的结果（匹配的位置，每个捕获分组匹配的内容等）都会计算出来，封装到`Match`对象中。访问`Match`对象的属性和方法，包括它的`Group`对象（及其属性和方法）只是取回已经计算好的结果。

核心对象详解

Core Object Details

概览完毕，来看细节。首先，我们来看如何创建`Regex`对象，然后来看如何将其应用到字符串，生成`Match`对象，以及如何处理这个`Match`对象和它的`Group`对象。

在实践中，很多时候不必明确创建`Regex`对象，不过明确创建看起来更顺眼，所以在讲解核心对象时，每次都会创建它们。稍后我会告诉你.NET提供的简便方法。

在下面的列表中，我会忽略从`Object`类继承而来的，很少用到的方法。

创建 Regex 对象

Creating Regex Objects

Regex 的构造函数并不复杂。它可以接收一个参数（作为正则表达式的字符串），或者是两个参数（一个正则表达式和一组选项）。下面是一个参数的例子：

```
Dim StripTrailWS = new Regex("\s+$") ' 去掉结尾的空白字符
```

它只是创建 Regex，做好应用前的准备；而没有进行任何匹配。

下面是使用两个参数的例子：

```
Dim GetSubject = new Regex("^subject: (.*)", RegexOptions.IgnoreCase)
```

这里多出了一个 RegexOptions 选项，不过可以用 OR 运算符连接多个选项，例如：

```
Dim GetSubject = new Regex("^subject: (.*)", _
                 RegexOptions.IgnoreCase OR RegexOptions.Multiline)
```

捕获异常

如果正则表达式包含了元字符的非法组合，就会抛出 ArgumentException。通常，如果用户知道所使用的正则表达式能够正常工作，就不需要捕获这个异常，不过如果使用程序“之外”（例如由用户输入，或者从配置文件读入）的正则表达式，就必须捕获这个异常。

```
Dim R As Regex
Try
    R = New Regex(SearchRegex)
Catch e As ArgumentException
    Console.WriteLine("*ERROR* bad regex: " & e.ToString)
    Exit Sub
End Try
```

显然，根据情况的不同，在检测到异常之后可能需要不同的处理：你可能需要进行其他的处理，而不仅仅是向控制台输出报错信息。

Regex 选项

在创建 Regex 对象时，可以使用下面的选项：

RegexOptions.**IgnoreCase**

　此选项表示，在应用正则表达式时，不区分大小写（☞110）。

RegexOptions.**IgnorePatternWhitespace**

　此选项表示，正则表达式应该按照自由格式和注释模式（☞111）来解析。如果使用单纯的「#…」注释，请确认在每一个逻辑行的末尾都有换行符，否则第一处注释会“注释掉”之后的整个正则表达式。

在 VB.NET 中，我们可以用 chr(10) 来实现，例如：

```
Dim R as Regex = New Regex( _
   "# 匹配一个浮点数 ...                              " & chr(10) & _
   " \d+(?:\.\d*)?     #  开头是整数部分...           " & chr(10) & _
   " |                 #  或者 ...                    " & chr(10) & _
   " \.\d+             #  开头是小数点", _
   RegexOptions.IgnorePatternWhitespace)
```

这样很累赘；VB.NET 提供了更简便的「(?#…)」注释：

```
Dim R as Regex = New Regex( _
   "(?#匹配一个浮点数...                            )" & _
   " \d+(?:\.\d*)?     (?#开头是整数部分...         )" & _
   " |                 (?#或者...                   )" & _
   " \.\d+             (?#开头是小数点              )", _
   RegexOptions.IgnorePatternWhitespace)
```

RegexOptions.**Multiline**

此选项表示，正则表达式在应用时应采用增强的行锚点模式（☞112）。也就是说，「^」和「$」能够匹配字符串内部的换行符，而不仅仅是匹配整个字符串的开头和结尾。

RegexOptions.**Singleline**

此选项表示，正则表达式使用点号通配模式（☞111）。此时点号能够匹配任意字符，也包括换行符。

RegexOptions.**ExplicitCapture**

此选项表示，普通括号「(…)」，在正常情况下是捕获型括号，但此时不捕获文本，而是与「(?:…)」一样，只分组，不捕获。此时只有命名捕获括号「(?<*name*>…)」能够捕获文本。

如果使用了命名分组，又希望使用非捕获型括号来分组，就可以使用正常的「(…)」括号和此选项，这样程序看起来更清晰。

RegexOptions.**RightToLeft**

此选项表示，进行从右向左的匹配（☞411）。

RegexOptions.**Compiled**

此选项表示，正则表达式应该在实际应用时被编译，成为高度优化的格式，这样通常会大大提高匹配速度。不过这样会增加第一次使用时的编译时间，以及程序执行期间的内存占用。

如果正则表达式只需要应用一次，或者应用并不是很频繁，就没必要使用 RegexOptions.Compiled，因为即使这个 Regex 对象已经被回收，多占的内存也不会释放。不过如果正则表达式在对时间要求很高的场合应用，这个选项可能非常有价值。

在第 237 页的例子中，使用这个选项减少了大约一半的测试时间。还可以参考关于编译到装配件（assembly）的讨论（☞434）。

RegexOptions.**ECMAScript**

此选项表示，正则表达式应该按照 ECMAScript（☞412）兼容方式来解析。如果不清楚 ECMAScript，或者不需要兼容它，可以直接忽略。

RegexOptions.**None**

它表示“没有额外的选项”，在初始化 RegexOptions 变量时，如果需要指定选项，可以使用它。也可以用 OR 来连接其他希望使用的选项。

使用 Regex 对象

Using Regex Objects

在没有实际应用之前，Regex 是没有意义的，下面的程序示范了实际的应用：

***RegexObj*.IsMatch(*target*)**   Return type: **Boolean**
***RegexObj*.IsMatch(*target*, *offset*)**

IsMatch 方法把目标正则表达式应用到目标字符串，返回一个 Boolean 值，表示匹配尝试是否成功，这里有个例子：

```
Dim R as RegexObj = New Regex("^\s*$")
  ......
If R.IsMatch(Line) Then
   ' 如果行为空 ...
     ......
Endif
```

如果提供了 *offset*（一个整数），则第一次尝试会从对应的偏移值开始。

***RegexObj*.Match (*target*)**   Return type: **Match** object
***RegexObj*.Match (*target*, *offset*)**
***RegexObj*.Match (*target*, *offset*, *maxlength*)**

Match 方法把正则表达式应用到目标字符串中，返回一个 Match 对象。通过这个 Match 对象可以查询匹配结果的信息（是否匹配成功，捕获的文本等等），初始化此正则表达式的“下一次”匹配。Match 对象的细节见第 427 页。

如果提供了 *offset*（一个整数），则第一次尝试会从对应的偏移值开始。

如果提供了 *maxlength* 参数，会进行特殊模式的匹配，从 *offset* 开始的字符开始计算，正则引擎会把 *maxlength* 长度的文本当作整个目标字符串，假设此范围之外的字符都不存在，所

以此时「^」能够匹配原来的目标字符串中的 *offset* 位置，「$」能够匹配之后 *maxlength* 个字符的位置。同样，环视结构不能“感觉到”此范围之外的字符。这与提供 *offset* 有很大不同，如果只提供了 *offset*，受影响的只是传动装置开始应用正则表达式的位置——正则引擎仍然能够“看到”完整的目标字符串。

下面表格中的例子比较了 *offset* 和 *maxlength* 的意义：

调用方法	以下列表达式 *RegexObj* 的结果		
	「\d\d」	「^\d\d」	「^\d\d$」
RegexObj.Match("May 16,1998")	匹配‘16’	失败	失败
RegexObj.Match("May 16,1998", 9)	匹配‘99’	失败	失败
RegexObj.Match("May 16,1998", 9, 2)	匹配‘99’	匹配‘99’	匹配‘99’

RegexObj.**Matches** (target)　　Return type: **MatchCollection**
RegexObj.**Matches** (target, offset)

Matches 方法类似 Match 方法，只是 Matches 方法返回一组 Match 对象，代表目标字符串中的**所有**匹配结果，而不是**第一次**的匹配结果。返回的对象为 MatchCollection。

例如，初始化代码如下：

```
Dim R as New Regex("\w+")
Dim Target as String = "a few words"
```

下面的程序：

```
Dim BunchOfMatches as MatchCollection = R.Matches(Target)
Dim I as Integer
For I = 0 to BunchOfMatches.Count - 1
    Dim MatchObj as Match = BunchOfMatches.Item(I)
    Console.WriteLine("Match: " & MatchObj.Value)
Next
```

运行结果是：

```
Match: a
Match: few
Match: words
```

下面的程序输出同样的结果，它说明，MatchCollection 对象可以一次分配整个 MatchCollection。

```
Dim MatchObj as Match
For Each MatchObj in R.Matches(Target)
    Console.WriteLine("Match: " & MatchObj.Value)
Next
```

作为比较，下面的代码也可以达到同样的效果，使用 Match（而不是 Matches）方法：

```
Dim MatchObj as Match = R.Match(Target)
While MatchObj.Success
    Console.WriteLine("Match: " & MatchObj.Value)
    MatchObj = MatchObj.NextMatch()
End While
```

RegexObj.**Replace** (*target*, *replacement*)　　　Return type: **String**
RegexObj.**Replace** (*target*, *replacement*, *count*)
RegexObj.**Replace** (*target*, *replacement*, *count*, *offset*)

Replace 方法会在目标字符串中进行查找-替换，返回（有可能已经变化的）字符串副本。它应用的是 Regex 对象的正则表达式，返回的不是 Match 对象，而是替换的结果。匹配的文本被什么内容替换，取决于 *replacement* 参数。*replacement* 参数可以重载：它可以是一个字符串，也可以是 MatchEvaluator 委托（delegate）。如果 *replacement* 是一个字符串，它会按照下一页补充内容的说明进行处理。例如：

```
Dim R_CapWord as New Regex("\b[A-Z]\w*")
  ……

Text = R_CapWord.Replace(Text, "<B>$0</B>")
```

把每一个大写单词两边加上<B>…</B>。

如果设置了 *count*，就只会进行 *count* 次替换（默认情况是进行所有的替换）。如果只希望替换第一次匹配，可以将 *count* 设置为 1。如果我们知道只会有一次匹配，把 *count* 明确设置为 1 的效率会更高，因为不需要对字符串的其他部分进行查找和处理。把 *count* 设置为-1 表示“所有匹配都必须替换”（它等价于没有设置 *count*）。

如果设置了 *offset*（一个整数），则应用正则表达式时，目标字符串中对应数目的字符会被忽略。这些忽略的字符会直接被复制到结果中。

例如，这段代码会去掉多余的空白字符（也就是将连续的多个空白字符替换为单个空格）：

```
Dim AnyWS as New Regex("\s+")
  ……

Target = AnyWS.Replace(Target, " ")
```

‘some·····random·····spacing’ 被替换为 ‘some·random·spacing’。下面代码的结果相同，只是它会保留行开头任意数目的空白字符。

```
Dim AnyWS     as New Regex("\s+")
Dim LeadingWS as New Regex("^\s+")
  ……

Target = AnyWS.Replace(Target, " ", -1, LeadingWS.Match(Target).Length)
```

它会把‘····some···random·····spacing’转化为‘····some·random·spacing’，在查找和替换时，它使用 LeadingWS 匹配文本的长度作为偏移值（就是要跳过的字符数目）。这里用到了 Match 对象的简便特性，即 LeadingWS.Match(Target)的 Length 属性（即便失败也没问题，此时 Length 的值为 0，也就是说我们需要对整个目标字符串应用 AnyWS）。

特殊的 Replacement 处理

Regex.Replace 方法和 Match.Result 方法都可以接收能够进行特殊处理的 replacement 字符串。下面的字符序列会被匹配的文本所替换：

字符序列	替换内容
$&	整个表达式匹配的文本（等于$0）
$1, $2 …	对应编号的捕获分组匹配的文本
${name}	对应命名捕获分组匹配的文本
$`	目标字符串中匹配文本**之前**的文本
$'	目标字符串中匹配文本**之后**的文本
$$	单个‘$’字符
$_	整个原始目标字符串的副本
$+	见说明

目前，$+几乎是没有用的。它源自 Perl 中的$+变量，即**实际参与**匹配的捕获型括号中编号最大的那个（第 202 页有一个例子）。而.NET 的 replacement 字符串中的$+只是引用正则表达式中最靠后的那个捕获型括号匹配的文本。因为捕获型括号会重新编号——在使用命名型捕获时，会自动进行这种操作（☞409），所以它基本没有用。

除了上面的情况之外，任何情况下在 replacement 字符串中使用‘$’都完全没问题。

使用 replacement 委托

replacement 参数不只能用简单字符串，还可以是委托（*delegate*，简单说就是函数指针）。代理函数在每次匹配之后调用，生成作为 replacement 的文本。因为这个函数能够进行我们需要的任何处理，这种查找替换的机制功能非常强大。

委托的类型是 MatchEvaluator，每次匹配都会调用。它所引用的函数必须接受 Match 对象，进行你所需要的任何处理，返回作为 replacement 的文本。

做个比较，下面两段程序输出同样的结果：

```
Target = R.Replace(Target, "<<$&>>"))
---------------------------------
Function MatchFunc(ByVal M as Match) as String
  return M.Result("<<$&>>")
End Function
Dim Evaluator as MatchEvaluator = New MatchEvaluator(AddressOf MatchFunc)
   ......

Target = R.Replace(Target, Evaluator)
```

两段程序都用<<…>>标注匹配的文本。使用委托的好处在于，在计算 replacement 时我们可以进行任意复杂的操作。下面的例子把摄氏温度转换为华氏温度：

```
Function MatchFunc(ByVal M as Match) as String
  '从$1 获得温度数值，转换为华氏温度
  Dim Celsius as Double = Double.Parse(M.Groups(1).Value)
  Dim Fahrenheit as Double = Celsius * 9/5 + 32
  Return Fahrenheit & "F" '添加"F"，然后返回
End Function

Dim Evaluator as MatchEvaluator = New MatchEvaluator(AddressOf MatchFunc)
......

Dim R_Temp as Regex = New Regex("(\d+)C\b", RegexOptions.IgnoreCase)
Target = R_Temp.Replace(Target, Evaluator)
```

如果目标字符串中包含‘`Temp is 37C.`’，它会被替换为‘`Temp is 98.6F.`’。

***RegexObj*.Split (*target*)**　　Return type: array of **String**
***RegexObj*.Split (*target*, *count*)**
***RegexObj*.Split (*target*, *count*, *offset*)**

`Split` 方法将目标正则表达式应用于目标字符串，返回由各匹配分隔的字符串数组。如下面这个例子所示：

```
Dim R as New Regex("\.")
Dim Parts as String() = R.Split("209.204.146.22")
```

`R.Split` 返回包含四个字符串的数组（‘209’、‘204’、‘146’和‘22’），它们由「\.」在目标字符串中的三次匹配来分隔。

如果提供了 *count* 参数，则至多返回 *count* 个字符串（除非使用了捕获型括号，一会儿会说到这个问题）。如果没有提供 *count*，`Split` 返回所有匹配分隔的字符串。提供 *count* 的意思是，正则表达式可能在找到最终匹配之前停止应用，若果真如此，数组中最后的元素就是目标字符串中余下的部分。

```
Dim R as New Regex("\.")
Dim Parts as String() = R.Split("209.204.146.22", 2)
```

此时，`Parts` 得到两个字符串，‘209’和‘204.146.22’。

如果设置了 *offset*（一个整数），则正则表达式的匹配尝试从对应编号的字符开始。前面的 *offset* 个字符会作为数组的第一个元素返回（如果设置了 `RegexOptions.RightToLeft`，就会作为最后一个元素）。

在 Split 中使用捕获型括号

如果出现了任何形式的捕获型括号，数组中通常会包含额外的捕获文本（也有些情况下根本不会包含）。来看个简单的例子，要拆分字符串‘`2006-12-31`’或是‘`04/12/2007`’，你可能会使用「`[-/]`」：

```
Dim R as New Regex("[-/]")
Dim Parts as String() = R.Split(MyDate)
```

结果包含 3 个元素（均为字符串）。不过，使用捕获型括号的正则表达式「`([-/,])`」，则会返回 5 个字符串：如果 `MyDate` 包含‘`2006-12-31`’，这 5 个元素是‘`2006`’、‘`-`’、‘`12`’、‘`-`’、‘`31`’。多出来的‘`-`’是每次捕获的`$1`。

如果有多组捕获型括号，它们会按照编号排序（也就是说，所有的命名捕获跟随在未命名捕获之后☞409）。

只要实际参与了匹配捕获型括号的捕获型括号，都会包含在 `Split` 的结果中。不过，目前的.NET 有一个 bug，即如果某组捕获型括号没有参与匹配，它和所有编号更靠后的捕获型括号都不会包含在返回的结果中。

来看个极端点的例子，如果需要以左右可能出现空白字符的逗号作为分隔，而且空白字符必须包含在返回结果中。用「`(\s+)?,(\s+)?`」分隔‘`this·,··that`’，得到四个字符串‘`this`’、‘`·`’、‘`··`’和‘`that`’。但是，如果目标字符串为‘`this,·that`’，因为第一组捕获型括号没有参与最终匹配，所有的捕获型括号都不包含在最终结果中，所以只会返回两个字符串‘`this`’和‘`that`’。无法预知到底会返回多少字符串，是当前版本的.NET 的一个重大问题。

在这个例子中，我们可以使用「`(\s*),(\s*)`」绕开这个问题（这样两个分组一定都能参与匹配）。不过，更复杂的表达式就没这么容易改写了。

RegexObj.**GetGroupNames()**
RegexObj.**GetGroupNumbers()**
RegexObj.**GroupNameFromNumber(** *number* **)**
RegexObj.**GroupNumberFromName(** *name* **)**

这几个方法容许用户查询对应编号（可以用数字，如果是命名捕获，也可以用名字）的捕获型分组的信息。它们引用的不是特定的匹配内容，只是正则表达式中存在的分组的名字和编号。下面的补充内容说明了使用方法。

RegexObj.**ToString()**
RegexObj.**RightToLeft**
RegexObj.**Options**

这几个方法容许用户查询 Regex 对象本身（而不是将此对象应用到字符串上）的信息。ToString() 方法返回正则表达式构造函数接收的字符串。RightToLeft 属性返回一个 Boolean 值，表明它是否启用了 RegexOptions.RightToLeft 选项。Options 属性返回与此正则表达式相关的 RegexOptions。下面说明了各个选项的值，把对应选项的值相加，就得到返回结果。

```
0  None                16   Singleline
1  IgnoreCase          32   IgnorePatternWhitespace
2  Multiline           64   RightToLeft
4  ExplicitCapture    256   ECMAScript
8  Compiled
```

这里没有 128，因为它用于微软内部的调试，没有出现在最终产品中。

补充内容给出了这些方法的应用实例。

使用 Match 对象

Using Match Objects

有三种方法创建 Match 对象：Regex 的 Match 方法、静态函数 Regex.Match（稍后介绍）和 Match 对象自己的 NextMatch 方法。它封装某个正则表达式的单次应用的所有相关信息。其属性和方法如下：

MatchObj.**Success**

返回一个 Boolean 值，表示匹配是否成功。如果不成功，则返回一个静态的 Match.Empty 对象（☞433）。

MatchObj.**Value**
MatchObj.**ToString()**

它返回实际匹配文本的副本。

显示 Regex 对象的信息

这段代码显示了 Regex 对象 R 的信息

```
'显示关于 Regex 变量 R 的已知信息
Console.WriteLine("Regex is: " & R.ToString())
Console.WriteLine("Options are: " & R.Options)
If R.RightToLeft
   Console.WriteLine("Is Right-To-Left: True")
Else
   Console.WriteLine("Is Right-To-Left: False")
End If

Dim S as String
For Each S in R.GetGroupNames()
    Console.WriteLine("Name """ & S & """ is Num #" & _
                      R.GroupNumberFromName(S))
Next
Console.WriteLine("---")
Dim I as Integer
For Each I in R.GetGroupNumbers()
    Console.WriteLine("Num #" & I & " is Name """ & _
                      R.GroupNameFromNumber(I) & """")
Next
```

再执行一次，用下面的方法创建 Regex 对象：

```
New Regex("^(\w+)://([^/]+)(/\S*)")
New Regex("^(?<proto>\w+)://(?<host>[^/]+)(?<page>/\S*)",
           RegexOptions.Compiled)
```

得到下面的结果（为适应排版，有个正则表达式省略了后半段）

```
Regex is: ^(\w+)://([^/]+)(/\S*)
Option are: 0
Is Right-To-Left: False
Name "0" is Num #0
Name "1" is Num #1
Name "2" is Num #2
Name "3" is Num #3
---
Num #0 is Name "0"
Num #1 is Name "1"
Num #2 is Name "2"
Num #3 is Name "3"
```

```
Regex is: ^(?<proto>\w+)://(?<host> …
Option are: 8
Is Right-To-Left: False
Name "0" is Num #0
Name "proto" is Num #1
Name "host" is Num #2
Name "page" is Num #3
---
Num #0 is Name "0"
Num #1 is Name "proto"
Num #2 is Name "host"
Num #3 is Name "page"
```

MatchObj.**Length**

返回实际匹配文本的长度。

MatchObj.**Index**

返回一个整数，显示匹配文本在目标中的起始位置。编号从 0 开始，所以这个数字表示从目标字符串的开头（最左边）到匹配文本的开头（最左边）的长度。即使在创建 Match 对象时设置了 RegexOptions.RightToLeft，回值也不会变化。

MatchObj.**Groups**

此属性是一个 GroupCollection 对象，其中封装了多个 Group 对象。它是一个普通的集合类（collection），包含了 Count 和 Item 属性，但是最常用的办法还是按照索引值访问，取出对应的 Group 对象。例如，M.Groups(3)对应第 3 组捕获型括号，M.Groups("HostName")对应命名捕获“HostName”（正则表达式中的「(?<HostName>…)」）。

在 C#中，使用 M.Groups[3]和 M.Groups["HostName"]。

编号为 0 的分组表示整个正则表达式匹配的所有文本。*MatchObj*.Groups(0).Value 等价于 *MatchObj*.Value。

MatchObj.**NextMatch()**

NextMatch()方法将正则表达式应用于目标字符串，寻找下一个匹配，返回新的 Match 对象。

MatchObj.**Result(*string*)**

string 是一个特殊的序列，按照第 424 页补充内容的介绍来处理，返回结果文本。这里有个简单例子：

```
Dim M as Match = Regex.Match(SomeString, "\w+")
Console.WriteLine(M.Result("The first word is '$&'"))
```

下面的程序可以依次匹配内容左侧和右侧文本的副本

```
M.Result("$`")  '这是匹配内容左侧的文本
M.Result("$'")  '这是匹配内容右侧的文本
```

调试时可能需要显示某些和行有关的信息：

```
M.Result("[$`<$&>$']"))
```

如果把「\d+」应用到‘May 16, 1998’得到的 Match 对象，返回的是‘May <16>, 1998’，这清楚地体现了匹配文本。

MatchObj.**Synchronized()**

它返回一个新的，与当前 `Match` 完全一样的 `Match` 对象，只是它适合于多线程使用。

MatchObj.**Captures**

`Captures` 属性并不常用，参见第 437 页的介绍。

使用 Group 对象

Using Group Objects

`Group` 对象包含一组捕获型括号（如果编号是 0，就表示整个匹配）的信息。其属性和方法如下：

GroupObj.**Success**

它返回一个 Boolean 值，表明此分组是否参与了匹配。并不是所有的分组都必须“参与”成功的全局匹配。如果「`(this)|(that)`」能够成功匹配，肯定有一个分组能参与匹配，另一个不能。第 139 页的脚注中有另一个例子。

GroupObj.**Value**
GroupObj.**ToString()**

它们都返回本分组捕获文本的副本。如果匹配不成功，则返回空字符串。

GroupObj.**Length**

返回本分组捕获文本的长度。如果匹配不成功，则返回 0。

GroupObj.**Index**

返回一个整数，表示本分组捕获的文本在目标字符串中的位置。编号从 0 开始，所以它就是从目标字符串的开头（最左边）到捕获文本的开头（最左边）的长度（即使在创建 `Match` 对象时设置了 `RegexOptions.RightToLeft`，结果仍然不变）。

GroupObj.**Captures**

请参考第 437 页 `Group` 对象的 `Capture` 属性。

静态“便捷”函数

Static “Convenience” Functions

在第 413 页的“正则表达式快速入门”中已经看到，我们并不需要每次都显式地创建 Regex 对象。我们可以通过下面的静态函数直接使用正则表达式：

```
Regex.IsMatch(target, pattern)
Regex.IsMatch(target, pattern, options)

Regex.Match(target, pattern)
Regex.Match(target, pattern, options)

Regex.Matches(target, pattern)
Regex.Matches(target, pattern, options)

Regex.Replace(target, pattern, replacement)
Regex.Replace(target, pattern, replacement, options)

Regex.Split(target, pattern)
Regex.Split(target, pattern, options)
```

其实它们不过是包装了已经介绍的主要的 Regex 构造函数和方法而已。它们会临时创建一个 Regex 对象，用它来调用请求的方法，然后弃用这个对象（其实并没有弃用，稍后介绍）

这里有个例子：

```
If Regex.IsMatch(Line, "^\s*$")
  ......
```

它等价于：

```
Dim TemporaryRegex = New Regex("^\s*$")
If TemporaryRegex.IsMatch(Line)
  ......
```

或者，更确切地说是：

```
If New Regex("^\s*$").IsMatch(Line)
  ......
```

使用这些便捷函数的好处在于，代码因此更清晰易懂。面向对象式处理看起来像函数式处理（☞95），坏处在于每次调用都必须重新检查 *pattern* 字符串。

如果在整个程序的执行过程中，正则表达式只用到 1 次，就不需要考虑便捷函数的效率问题。但是，如果需要应用多次（例如在循环中，或者是频繁调用的函数中），每次准备正则表达式都需要代价（☞241）。创建 Regex 对象，然后重复使用的主要原因之一就是，使用便捷函数的效率太低。不过，下一节将告诉我们，.NET 提供了一种很好的解决办法：兼具面向对象的效率和函数式处理的便捷。

正则表达式缓存

Regex Caching

为简单的正则表达式构建并管理 Regex 对象很不方便，所以.NET 的正则包提供了各种静态方法。但这些静态方法存在效率缺陷，即每次调用都需要创建临时的 Regex 对象，应用它，然后弃用。如果在循环中需要多次应用同样的正则表达式，就需要进行许多不必要的工作。

为了避免重复的工作，.NET Framework 能够缓存静态方法创建的临时变量。第 6 章已经大致介绍过缓存（☞244），简单地说，缓存的意思就是如果你希望在静态方法中使用"最近"使用过的正则表达式，此方法就会重用之前创建的正则对象，而不是重新创建一个新对象。

"最近"的默认意义是缓存 15 个正则表达式。如果循环中使用的正则表达式超过 15 个，则第 16 个正则表达式会取代第 1 个，所以进入下一轮循环时，第一个正则表达式已经不在缓存中，必须重新生成。

如果默认值 15 太小，可以这样调整：

```
Regex.CacheSize = 123
```

如果希望禁用缓存，可以将其设置为 0。

支持函数

Support Functions

除了之前讨论过的便捷函数，还有一些静态的支持函数：

Regex.Escape(*string*)

Regex.Escape(…)返回此字符串的副本，其中的元字符会进行转义。这样处理的字符串就能够作为文字字符串供正则表达式使用。

例如，如果用户的输入保存在字符串 SearchTerm 中，我们可以这样构建正则表达式：

```
Dim UserRegex as Regex = New Regex("^" & Regex.Escape(SearchTerm) & "$", _
                                   RegexOptions.IgnoreCase)
```

这样，用户输入的元字符就不会被特殊处理了。如果不转义，假设用户输入了'`:-)`'，就会抛出 ArgumentException 异常（☞419）。

Regex.Unescape(*string*)

这个函数有点奇怪，它接收一个字符串，返回此字符串的副本，不过要处理其中的元字符，去掉其他的反斜线。如果输入的是‘\:\-\)’，返回值就是‘:-)’。

字符缩略表示法也会被解码。接收的字符串中的‘\n’会被替换为换行符，‘\u1234’会被替换为对应的 Unicode 字符。第 407 页列出的所有字符缩略表示法都会被处理。

我想象不出 Regex.Unescape 有多么重要的价值，不过了解转义规定的用户也许能把它当作生成 VB 字符串的通用工具。

Match.Empty

此函数返回代表匹配失败的 Match 对象。它的用处可能在于，如果初始化的某个 Match 对象将来不一定会被用到，但又必须能够查询。这里有个简单的例子：

```
Dim SubMatch as Match = Match.Empty '初始化一个Match，但下面不一定会设定
  ......

Dim Line as String
For Each Line in EmailHeaderLines
   '如果这是标题，保存匹配的信息 ...
   Dim ThisMatch as Match = Regex.Match(Line, "^Subject:\s*(.*)", _
                                        RegexOptions.IgnoreCase)
   If ThisMatch.Success
      SubMatch = ThisMatch
   End If
      ......
Next
  ......

If SubMatch.Success
   Console.WriteLine(SubMatch.Result("The subject is: $1"))
Else
   Console.WriteLine("No subject!")
End If
```

如果字符串数组 EmailHeaderLines 没有任何行（或者没有 Subject 行），程序中的循环就不会设置 SubMatch，如果 SubMatch 没有初始化，循环之后检查 SubMatch 会得到一个空引用异常。这种情况下用 Match.Empty 来初始化就很方便。

Regex.CompileToAssembly(…)

它容许用户创造一个装配件（assembly），封装正则表达式——参见下一节。

.NET 高级话题

Advanced .NET

下面的内容涉及某些尚未介绍过的特性：通过正则装配件（regex assemblies）构建正则表达式库，使用.NET 专属的特性匹配嵌套结构，以及对 Capture 对象的讲解。

正则表达式装配件

Regex Assemblies

.NET 能够把 Regex 对象封装到装配件（assembly）中，在构建正则表达式库时这很有用。下一页的补充内容提供了示例。

运行补充内容中的例子，能够在当前工程的 *bin* 代码目录下创建 *JfriedlsRegexLibrary. DLL*。

然后我们可以通过 Visual Studio .NET 的 *Project > Add Reference* 将其加入，在其他工程中使用这个装配件。

要使用装配件中的类，首先必须导入：

```
Imports jfriedl
```

然后就可以像其他任何类一样引用它们，例如：

```
Dim FieldRegex as CSV.GetField = New CSV.GetField '生成新的 regex 对象
  ......

Dim FieldMatch as Match = FieldRegex.Match(Line) '应用到字符串 ...
While FieldMatch.Success
  Dim Field as String
  If FieldMatch.Groups(1).Success
    Field = FieldMatch.Groups("QuotedField").Value
    Field = Regex.Replace(Field, """""", """") '把连在一起的引号替换为单个引号
  Else
    Field = FieldMatch.Groups("UnquotedField").Value
  End If

  Console.WriteLine("[" & Field & "]")
  ' 现在可以处理' Field'...

  FieldMatch = FieldMatch.NextMatch
End While
```

在这个例子中，我仅仅从 jfriedl namespace 导入，但也可以很简单地从 jfiedl.CSV namespace 导入，然后这样创建 Regex 对象：

```
Dim FieldRegex as GetField = New GetField '生成新的 regex 对象
```

这是两种风格。

通过装配件构建自己的正则表达式库

这个例子构建了一个小规模的正则表达式库。完整的程序构建了一个装配件（DLL），其中包含三个已经生成的 Regex 构造函数：jfriedl.Mail.Subject，jfriedl.Mail.From 和 jfriedl.CSV.GetField。

前两者很简单，一眼就能明白，最后那个复杂的构造函数展示了构建库的约定（promise）。请注意，这里不需要设定 RegexOptions.Compiled，因为在构造装配件时已经隐含地设定了。

关于如何使用构建好的装配件，请参考第 434 页。

```
Option Explicit On
Option Strict On

Imports System.Text.RegularExpressions
Imports System.Reflection

Module BuildMyLibrary
Sub Main()
 ' 下面调用 regexCompliationInfo 时提供了 pattern、regex 选项、类内部的名称、类名，
 ' 以及一个 Boolean 值表明类是否 public。举例来说，要使用第一个类，
 ' 程序必须使用装配件中"jfriedl.Mail.Subject"作为 Regex 的构造函数。
Dim RCInfo() as RegexCompilationInfo = {                                    _
  New RegexCompilationInfo(                                                 _
      "^Subject:\s*(.*)", RegexOptions.IgnoreCase,                          _
      "Subject", "jfriedl.Mail", true),                                     _
  New RegexCompilationInfo(                                                 _
        "^From:\s*(.*)", RegexOptions.IgnoreCase,                           _
        "From", "jfriedl.Mail", true),                                      _
  New RegexCompilationInfo(                                                 _
       "\G(?:^|,)                                            " &            _
       "(?:                                                  " &            _
       " (?# 或者是双引号字段... )                           " &            _
       " "" (?# 起始双引号 )                                 " &            _
       " (?<QuotedField> (?> [^""]+ | """" )* )              " &            _
       " "" (?# 结束双引号 )                                 " &            _
       " (?# ...或者... )                                    " &            _
       " |                                                   " &            _
       " (?# ...非引号/非逗号文本... )                       " &            _
       " (?<UnquotedField> [^"",]*)                          " &            _
       ")",                                                                 _
       RegexOptions.IgnorePatternWhitespace,                                _
       "GetField", "jfriedl.CSV", true)                                     _
  }
  '现在进行主要的处理，生成结果...
  Dim AN as AssemblyName = new AssemblyName()
  AN.Name = "JfriedlsRegexLibrary" 'DLL 文件的名字
  AN.Version = New Version("1.0.0.0")
  Regex.CompileToAssembly(RCInfo, AN)'构建完成
End Sub
End Module
```

也可以不进行任何导入，而是直接使用：

```
Dim FieldRegex as jfriedl.CSV.GetField = New jfriedl.CSV.GetField
```

这有点麻烦，但是清楚地说明了对象的出处。同样，这只是风格的问题。

匹配嵌套结构

Matching Nested Constructs

微软提供了一种创新的功能，专门用于匹配对称的结构（长期以来，正则表达式对此无能为力）。它理解起来并不容易——本节篇幅不长，但请注意，其中的内容分量不少。

最简单的办法就是用一个例子来说明：

```
Dim R As Regex = New Regex( " \(                      " & _
                            "   (?>                   " & _
                            "       [^()]+            " & _
                            "     |                   " & _
                            "       \( (?<DEPTH>)     " & _
                            "     |                   " & _
                            "       \) (?<-DEPTH>)    " & _
                            "   )*                    " & _
                            "   (?(DEPTH)(?!))        " & _
                            " \)                      ",  _
        RegexOptions.IgnorePatternWhitespace)
```

它匹配第一组正确配对的嵌套括号，例如‘before (nope (yes (here) okay) after’中用下画线标注的部分。第一个开括号不会匹配，因为它没有对应的闭括号。

这里简要说明了程序的工作原理：

1. 每匹配一个（超过正则表达式开头「\(」的）‘(’，「(?<DEPTH>)」会把正则表达式保存的当前括号嵌套深度值加 1。

2. 每匹配一个‘)’，「(?<-DEPTH>)」会把深度减 1。

3. 「(?(DEPTH)(?!))」确保最后的「\)」匹配时，深度应该为 0。

因为引擎的回溯堆栈保存了当前匹配成功分组的信息，这个办法没有问题。「(?<DEPTH>)」只是使用了命名捕获的「()」，它总是能成功匹配。因为它紧跟在「\(」之后，它的成功匹配（此信息会保存在堆栈中，直到出栈为止）用于标记开括号的数目。

因此，当前已经成功匹配的‘DEPTH’分组总数就保存在回溯堆栈中。我们希望在找到闭括号之后减去它们。.NET 独有的「(?<-DEPTH>)」结构，会从堆栈中去掉最近的“successful

DEPTH” 标记。如果不存在这样的标记，「(?<-DEPTH>)」就会报告失败，整个正则表达式的匹配宣告失败。

最后的「(?(DEPTH)(?!))」是一个普通的条件判断，如果‘DEPTH’分组匹配成功它会应用「(?!)」。如果在程序运行到此处时选择应用此分支，就表示还存在未匹配的开括号。果真如此的话，我们就需要退出匹配（我们不希望匹配不对称的序列）所以我们用否定型顺序环视「(?!)」来做检查，确保匹配失败。

看到了吗？这就是.NET 正则表达式匹配嵌套结构的原理。

Capture 对象

Capture Objects

.NET 的对象模型中还包括 Capture 对象，之前一直没有介绍过。依视角的不同，它可能为匹配结果增加了新的观察角度，也可能是增加把结果弄得更糟。

Capture 对象几乎等价于 Group 对象，因为它表示一组捕获型括号匹配的文本。与 Group 对象一样，它提供了 Value（匹配的文本）、Length（匹配文本的长度），以及 Index（匹配文本在目标字符串中的偏移值，编号从 0 开始）。

Group 对象和 Capture 对象的主要差别是，每个 Group 对象都包含了一组 Captures，分别对应到匹配过程中各分组的**未确定**匹配（intermediary match），以及该分组**最终**匹配的文本。

看下面这个例子：

```
Dim M as Match = Regex.Match("abcdefghijk", "^(..)+")
```

正则表达式匹配了 5 组「(..)」，包括了字符串中的绝大多数字符‘abcdefghijk’：因为加号在括号外面，加号控制的每次迭代都会重新捕获，这个捕获型括号最后保存的是‘ij’（也就是说，M.Groups(1).Value 等于‘ij’）。相反，M.Groups(1)同样包含一组 Capture，它们对应到「(..)」的匹配过程：

```
M.Groups(1).Captures(0).Value is  'ab'
M.Groups(1).Captures(1).Value is  'cd'
M.Groups(1).Captures(2).Value is  'ef'
M.Groups(1).Captures(3).Value is  'gh'
M.Groups(1).Captures(4).Value is  'ij'
M.Groups(1).Captures.Count is 5.
```

你也许会注意到，最后匹配的‘ij’等同于最终全局匹配中的M.Groups(1).Value。看起来，Group的Value就是本分组最终匹配文本的简记法。M.Groups(1).Value是：

```
M.Groups (1).Captures( M.Groups(1).Captures.Count - 1 ).Value
```

关于Capture，还要讲几点：

- M.Groups(1).Capture是一个CaptureCollection，与普通的集合类（collection）一样，它包含了Items和Count属性。不过，通常大家都不会使用这两个属性，而是通过索引值直接访问，例如M.Groups(1).Captures(3)（在C#中是M.Groups[1].Captures[3]）。
- Capture对象没有Success方法，如果需要，请测试Group的Success。
- 到现在，我们已经看到，Capture 对象在 Group 对象内部可用。Match 对象也有Captures 属性，尽管涌出并不大。M.Captures 可以直接访问编号为 0 的分组的Captures属性（也就是说M.Captures等价于M.Groups(0).Captures）。因为编号为0的分组表示整个匹配，所以不会有“遍历”匹配的迭代，所以编号为0的捕获集合类只有一个Capture。因为它们包含与编号为0的匹配同样的信息，M.Capters和M.Groups(0).Captures并不是很有用。

.NET的Capture对象是一种创新，但是因为与对象模型“集成过度（overly integrated)”，使用起来反而更复杂，而且令人迷惑。在仔细参阅了.NET的文档，并真正理解了这些对象之后，我感觉这种做法有利也有弊。一方面，我乐于看到这种创新。虽然它的用法并不会马上显现出来，但这或许是因为一直以来我都习惯于用传统的正则表达式特性来思考问题。

另一方面，在匹配过程中的额外的分组，匹配完成之后把它们封装到一个对象中，似乎降低了效率，我并不希望降低效率，除非要得到额外的信息。增加的Capture分组在大多数匹配中不会用到，但是照目前的情况来看，生成 Match 对象时会构建所有的 Group 和Capture对象（以及它们相关的GroupCollection和CaptureCollection对象)。所以无论是否需要，它们都在那里，如果你能够发现Capture对象的使用价值，就不要放过。

第10章 PHP

PHP

20 世纪 90 年代末期 Web 的迅猛发展导致了 PHP 的爆炸性流行，并一直持续至今。PHP 得以流行的理由之一是，即使非专业人员，只需要稍作准备，就能使用 PHP 的基本功能。除此之外，PHP 还提供了颇受开发老手欢迎的众多高级特性和函数。PHP 当然能够支持正则表达式，而且提供了至少 3 套独立的，不相关的正则引擎。

PHP 的三种正则引擎是“preg”、“ereg”和“mb_ereg”。本书介绍的是 **preg** 引擎提供的函数。它使用 NFA 引擎，通常情况下，在速度和功能方面都要强于其余两者（“preg”读作“p-reg”）。

与之前各章的联系 在开始本章之前，读者必须知道，本章的内容强烈依赖于第 1 至 6 章介绍的基础知识。如果读者只对 PHP 感兴趣，可能会直接从本章开始阅读，但我还是希望他们认真地看一看前言（尤其是关于体例）和前面的章节：第 1、2、3 章介绍了与正则表达式相关的基本概念、特性和技术，第 4、5、6 章介绍了一些理解正则表达式的关键知识，它们可以直接应用到 PHP 的正则表达式中。前几章讲解的重要概念包括 NFA 引擎进行匹配基本原理，匹配优先特性、回溯和效率。

接下来我要强调，除了用于速查列表——例如本章的第 407 页，和第 3 章从第 114 页到第 123 页，我并不希望这本书成为一本参考手册，而希望它成为精通正则表达式的详细教科书。



本章开始简要介绍了 preg 引擎的历史，然后介绍它的正则流派。接下来的几节详细考察了 preg 函数的接口，然后是关于 preg 的效率问题，最后是扩展示例。

preg 的背景和历史 preg 这个名字来自接口函数名的前缀，代表“Perl 的正则表达式（Perl Regular Expressions）”。preg 引擎的创始人是 Andrei Zmievski，他对当时作为标准的 ereg 套件的诸多掣肘相当不满意。（ereg 表示“扩展的正则表达式(extended regular expressions)”，它能兼容 POSIX 标准，“扩展”的意思是不仅仅限于一个最简单的正则流派，但是以今天的标准来看，还相当简陋）。

Andrei 的 preg 套件是一组 PCRE（即“Perl 兼容的正则表达式”Perl Compatible Regular Expressions）接口，这是一套非常棒的基于 NFA 的正则表达式库，完整地模拟了 Perl 的语法和语意，提供了 Andrei 想要的能力。

在接触 PCRE 以前，Andrei 先阅读了 Perl 的源代码，以决定是否能够借用到 PHP 当中。他显然不是第一个阅读 Perl 的正则表达式源代码的人，也不是第一个认识到代码有多么复杂的人。Perl 的正则表达式功能强，速度快，源代码也在许多年间经过了许多人的修改，已经超出了单个开发人员的理解能力。

幸运的是，剑桥大学的 Philip Hazel 同样已经被 Perl 的正则表达式的源代码搞得头昏脑胀，所以他写了 PCRE 库（参见第 91 页）。好在 Philip 已经有了现成的可供模拟的语意，所以若干年后，Andrei 找到了这套编写清晰、文档完备、效率出众的库，很方便就能将其绑定到 PHP 中。

Perl 随着时间的流逝而不断发展，PCRE 也是这样，PHP 亦然。本书针对的是 PHP Versions 4.4.3 和 5.1.4，这两者都兼容 PCRE Version6.6（注 1）。

如果你不熟悉 PHP 的版本信息，请注意 4.*x* 和 5.*x* 是同时维护的，而 5.*x* 进行了大规模的扩展。因为两个系列是分开维护和发布的，很可能某个 5.*x* 的版本所用的 PCRE 的版本要低于更晚发布的 4.*x* 版本。

注 1：在写作本章时，我研究了当时可用的 PHP 和 PCRE 的版本，发现了一些 bug，不过本书针对的 PHP 4.4.3 和 5.1.4 中已经修正了这些问题。在早期的版本中，本章的某些例子可能无法正常工作。

PHP 的正则流派

PHP's Regex Flavor

表 10-1：PHP preg 的正则流派

字符缩略表示法[1]	
☞115 (c)	`\a [\b] \e \f \n \r \t \octal\xhex \x{hex} \cchar`
字符组及相关结构	
☞118 (c)	字符组：`[…] [^…]`（可包含集合运算符☞125）
☞119	几乎任何字符：**点号**（根据模式的不同，有各种含义）
☞120 (u)	Unicode 混合序列：`\X`
☞120 (c)	字符组缩略表示法[3]：`\w \d \s \W \D \S`（只针对 8 位字符）[2]
☞121 (c) (u)	Unicode 属性和区块：`\p{Prop} \P{Prop}`
☞120	单个字节（可能有危险）[4]：`\C`
锚点及其他零长度断言	
☞129	行/字符串起始位置：`^ \A`
☞129	行/字符串结束位置[5]：`$ \z \Z`
☞130	当前匹配的起始位置：`\G`
☞133	单词分界符：`\b \B`（只针对 8 位字符）[2]
☞133	环视结构[6]：`(?=…) (?!…) (?<=…) (?<!…)`
注释及模式修饰符	
☞446	模式修饰符：`(?mods-mods)` 容许出现的模式：`x`[7] `s m i X U`
☞446	模式修饰范围：`(?mods-mods:…)`
☞136	注释：`(?#…)`（只在模式修饰符 x 下有效，从 '#' 到换行符或表达式末尾）
分组及捕获	
☞446	捕获型括号：`(…) \1 \2…`
☞446	命名捕获：`(?P<name>…) (?P=name)`
☞137	仅分组的括号：`(?:…)`
☞139	固化分组：`(?>…)`
☞139	多选结构：`|`
☞475	递归：`(?R) (?num) (?P>name)`
☞140	条件判断：`(?if then|else)`——"if" 部分可以是环视，`(num)` 或 `(name)`
☞141	匹配优先量词：`* + ? {n} {n,} {x,y}`
☞141	忽略优先量词：`*? +? ?? {n}? {n,}? {x,y}?`
☞142	占有优先量词：`*+ ++ ?+ {n}+ {n,}+ {x,y}+`
☞136 (c)	文字（非元字符）范围：`\Q…\E`

(c) ——可用于字符组内部　　　　①……⑦见说明

(u) ——只能与模式修饰符 u 连用

（本表同样适用于 PHP preg 函数所使用的正则表达式库 PCRE）

前页的表10-1简要介绍了preg引擎的正则流派。下面是补充说明：

① 只有在字符组内部，\b才表示退格符。在其他场合，\b匹配单词分界符（☞133）。

十进制转义只能使用两到三位八位数值。特殊的一位数「\0」序列匹配空字节（NUL byte）。

「\x*hex*」容许出现一到两位十六进制数字，而「\x{*hex*}」容许任意多个数字。请注意，大于\x{FF}的数值只能与模式修饰符u连用（☞447）。如果没有模式修饰符u，大于\x{FF}的值会导致正则表达式非法。

② 即使是在UTF-8模式下（通过模式修饰符u），单词分界符和字符组简记法，例如「\w」，也只对ASCII字符起作用。如果需要处理所有的Unicode字符，请使用「\pL」（☞121）代替「\w」，用「\pN」代替「\d」，用「\pZ」代替「\s」。

③ Unicode支持针对Unicode Version 4.1.0。

Unicode字母表（☞122）的支持不需要任何'Is'或者'In'前缀，例如「\p{Cyrillic}」。

PHP同时支持单字母或双字母Unicode属性，例如「\p{Lu}」、「\p{L}」，其中「\pL」作为单字母属性名（☞121）。而不支持「\p{Letter}」之类的长名称。

PHP也支持特殊的「\p{L&}」（☞121），以及「\p{Any}」（表示任意字符）。

④ 在默认情况下，preg套件的正则表达式是以字节为单位的，所以「\C」默认就等价于「(?s:.)」，由s修饰的点号。不过，如果使用了修饰符u，则preg套件就会以UTF-8字母为单位，也就是说，一个字符可能由6个字节组成。即使这样，「\C」仍然匹配单个字节。请参考第120页的注意事项。

⑤ 「\z」和「\Z」都能够匹配字符串的末尾，而「\Z」同样能够匹配最后的换行符。

「$」的意义取决于模式修饰符m和D（☞446）：如果没有设定任何修饰符，「$」等价于「\Z」（在字符串结尾的换行符，或者是字符串结尾）；如果使用了m，则它能够匹配内嵌的换行符，如果使用了模式修饰符D，它能够匹配「\z」（只有在字符串的结尾）。如果同时设置了m和D，则忽略D。

⑥ 逆序环视中使用的子表达式只能匹配固定长度的文本，除非顶层多选分支容许不同的固定长度（☞133）。

⑦ 模式修饰符 x（自由格式和注释）只能识别 ASCII 的空白字符，不能识别 Unicode 中的空白字符。

Preg 函数接口

The Preg Function Interface

PHP 正则引擎的处理方式完全是程序式的（☞95），包括表 10-2 顶端的 6 个函数，表格还列举了 4 个有用的函数，将在本章后面提到。

表 10-2：PHP Preg 函数概览

函　数	用　途
☞449 preg_match	测试正则表达式能否在字符串中找到匹配，并提取数据
☞453 preg_match_all	从字符串中提取数据
☞458 preg_replace	在字符串的副本中替换匹配的文本
☞463 preg_replace_callback	对字符串中的每处匹配文本调用处理函数
☞465 preg_split	将字符串切分为子串数组
☞469 preg_grep	选出数组中能/不能由表达式匹配的元素
☞470 preg_quote	转义字符串中的正则表达式元字符
下面四个函数在本章中开发完成，列在此处方便查询	
☞454 reg_match	类似 preg_match，但能识别出为参与匹配的括号
☞472 preg_regex_to_pattern	根据正则表达式字符串生成 preg pattern 字符串
☞474 preg_pattern_error	检查 preg pattern 字符串的语法错误
☞475 preg_regex_error	检查正则表达式字符串的语法错误

每个函数的具体功能都取决于参数的个数、标志位（flag），以及正则表达式所使用的模式修饰符。在深入细节之前我们先通过几个例子来看看 PHP 中的正则表达式的例子和处理方式。

```
/* 测试HTML tag是否<table> tag */
if (preg_match('/^<table\b/i', $tag))
  print "tag is a table tag\n";

-----------------------------------
/* 测试文本是否为整数 */
if (preg_match('/^-?\d+$/', $user_input))
  print "user input is an integer\n";
-----------------------------------
```

```
/* 从字符串中取出HTML title */
if (preg_match('{<title>(.*?)</title>}si', $html, $matches))
  print "page title: $matches[1]\n";

------------------------------------
/* 将字符串中的数值作为华氏温度，将其替换为摄氏温度 */
$metric = preg_replace('/(-?\d+(?:\.\d+)?)/e', /* pattern */
                       'floor(($1-32)*5/9 + 0.5)', /* 替换代码 */
                       $string);
------------------------------------
/* 从逗号分割值数据创建字符串数组 */
$values_array = preg_split('!\s*,\s*,!', $comma_separated_values);
```

最后的程序，如果输入 'Larry,·Curly,·Moe'，返回三个元素的数组：'Larry'，'Curly' 和 'Moe'。

"pattern" 参数
"Pattern" Arguments

所有 preg 函数的第一个参数都是 *pattern*，正则表达式包含在一对分隔符之内，可能还跟有模式修饰符。在上面的第一个例子中，pattern 参数是 '/<table\b/i'，也就是包含在一对斜线（分隔符）里头的「<table\b」，然后是模式修饰符 i（不区分大小写的匹配）。

PHP 单引号字符串

因为正则表达式很有可能包含反斜线，，所以在以字符串文字方式提供 pattern 参数时，最好使用 PHP 的单引号字符串。第 3 章介绍了 PHP 的字符串文字（☞103），简单地说，如果使用单引号字符串文本，正则表达式就可以省略许多额外的转义。PHP 的单引号字符串只有两个元序列，'\'' 和 '\\'，分别代表单引号和反斜线。

有一种转义需要特别注意，就是在正则表达式中使用「\\」匹配一个反斜线字符。在单引号字符串中，每个「\」都应表示为\\，所以「\\」就成了\\\\。四个反斜线才能匹配一个反斜线字符，这真神奇！

（473 页有反斜线繁复到极端的例子。）

举个具体的例子，用正则表达式匹配 Windows 系统中的分区名，例如 'C:\'。可以用正则表达式「^[A-Z]:\\$」，表示为单引号字符串文字就是 '^[A-Z]:\\\\$'。

第 5 章第 190 页有一个例子，「^.*\\」作为 pattern 字符串时应该写成‘/^.*\\\/’，使用 3 个反斜线。照此类推，我找到这几个例子：

```
print '/^.*\/';       输出 /^.*\/
print '/^.*\\/';      输出 /^.*\/
print '/^.*\\\/';     输出 /^.*\\/
print '/^.*\\\\/';    输出 /^.*\\/
```

头两个例子尽管方式不同，结果却是一样的。在第一个例子中，结尾的‘\/’对于单引号字符串文字并不是特殊文本，所以它就等于字符串的值。第二个例子中，‘\\’对于字符串文字来说有特殊含义，所以输出的字符串中出现单个‘\’。它与后面的字符（斜线）放在一起，得到与第一个例子同样的‘\/’。同样的道理，第三个和第四个例子也会得到同样的结果。

当然，你也可以使用 PHP 的双引号字符串文本，但是它们要麻烦许多。它们支持许多字符串元序列，这些在正则表达式字符串中都必须特殊处理。

分隔符

preg 引擎要求正则表达式两端必须有分隔符，因为设计者希望它看起来更像 Perl，尤其在模式修饰符的使用方法上更是如此。有的程序员觉得在正则表达式两端添加分隔符简直是多此一举，但是无论好还是不好，规定就是规定（第 448 也给出了一个“不好”的原因）。

常见的做法是使用斜线作为分隔符，不过我们还可以用除了字母、数字、反斜线和空白字符之外的任何 ASCII 字符做分隔符。最常见的是一对斜线，但两个‘!’和‘#’也很常见。

如果第一个分隔符表示“开(opening)”：

```
{ ( < [
```

对应的“闭”分隔符就是：

```
} ) > ]
```

如果使用这样“配对”的分隔符，分隔符就可能嵌套，所以‘((\d+))’也可以用作 pattern 字符串。其中，外面的括号是模式字符串分隔符，内部的括号属于分隔符之内的正则表达式。为了清晰起见，我会避免这种情况，使用简单易懂的‘/(\d+)/’。

pattern 字符串内部可以出现转义的分隔符，所以‘/<B>(.*?)<\/B>/i’并没有错，不过换一组分隔符可能看得更清楚，例如‘!<B>(.*?)</B>!i’使用‘!…!’作为分隔符，而‘{<B>(.*?)</B>}i’使用‘{…}’。

模式修饰符

在结束分隔符之后可以跟随多种模式修饰符（用 PHP 的术语来说，叫做 *pattern modifier*），在某些情况下，修饰符也可以出现在正则表达式内部，修饰模式的某些性质。我们已经在一些例子中看到过表示不区分大小写的模式修饰符 `i`。下面简要介绍模式修饰符：

修饰符	表达式中的写法	说　明
i	「(?i)」	☞110 忽略大小写
m	「(?m)」	☞112 增强的行锚点模式
s	「(?s)」	☞111 点号通配模式
x	「(?x)」	☞111 宽松排列和注释模式
u		☞447 以 UTF-8 读取正则表达式和目标字符串
X	「(?X)」	☞447 启用 PCRE“额外功能（extra stuff）”
e		☞459 将 replacement 作为 PHP 代码（只用于 preg_replace）
S		☞478 启用 PCRE 的“study”优化尝试
下面三个很少用到		
U	「(?U)」	☞447 交换「*」和「*?」的匹配优先含义
A		☞447 将整个匹配尝试锚定在起始位置（译注 1）
D		☞447「$」只能匹配 EOS，而不是 EOS 之前的换行符（如果使用了模式修饰符 m 则不会这样）

表达式内部的模式修饰符　在正则表达式内部，模式修饰符可以单独出现，来启用或停用某些特性（例如用「(?i)」来启用不区分大小写的匹配，用「(?-i)」来停用☞135）。此时，它们的作用范围持续到对应的结束括号，如果不存在，就持续到正则表达式的末尾。

他们也可以用作**模式修饰范围**（☞135），例如「(?i:…)」表示对此括号内的内容进行不区分大小写的匹配，「(?-sm:…)」表示在此范围内停用 s 和 m 模式。

正则表达式之外，结束分隔符之后的模式修饰符可以以任何顺序组织，下例中的‘si’表示同时启用不区分大小写和点号通配模式：

```
if (preg_match('{<title>(.*?)</title>}si', $html, $captures))
```

译注 1：不启动驱动过程。

PHP 特有的修饰符 列表最上端的 4 个模式修饰符属于标准修饰符，在第 3 章（☞110）已经讨论过。修饰符 **e** 只能在 preg_replace 中使用，详细的讨论见对应的小节（☞459）。

模式修饰符 u 告诉 preg 引擎，以 UTF-8 编码处理正则表达式和目标字符串。此模式修饰符不会修改数据，只是更改正则引擎处理数据的方式。默认（也就是未使用模式修饰符 u）的情况下，preg 引擎认为接收的数据都是 8 位编码的（☞87）。如果用户知道数据是 UTF-8 编码的，请使用此修饰符，否则请不要使用。在 UTF-8 编码中，非 ASCII 字符以多个字节来存储，使用 u 修饰符能够确保多个**字节**会被作为单个**字符**来处理。

模式修饰符 **X** 启用 PCRE 的“额外功能（extra stuff）”，目前它只有一个效果：如果出现了无法识别的反斜线序列，就报告错误。例如，默认情况下，「\k」在 PCRE 中没有特殊意义，就等价于「k」(因为这不是一个已知的元序列，所以反斜线会被忽略)。如果使用了模式修饰符 **X**，就会报告“unrecognized character follows \”。

未来版本的 PHP 可能包含更高版本的 PCRE，其中当前没有特殊意义的反斜线组合可能被赋予新的意义，所以为了保持未来的兼容性（以及一般可读性），最好是不要转义不需要的字母，除非它们现在有特殊意义。从这个意义上说，模式修饰符 **X** 意义重大，因为它可以发现这样的错误。

模式修饰符 **S** 调用 PCRE 的“study（研究）”特性，预先分析正则表达式，在某些顺利的情况下，在尝试匹配时速度会大大提升。本章中关于效率的内容将对此有介绍，请参考第 478 页。

剩下的模式修饰符实用价值不大，也不常用：

- 模式修饰符 **A** 把匹配锚定在第一次尝试的位置，就等于整个正则表达式以「\G」开头。如果用第 4 章的汽车的类比，这就是关闭传动机构的“驱动过程”（☞148）。
- 模式修饰符 **D** 会把每个「$」替换为「\z」(☞112)，即「$」匹配字符串的末尾，而不是字符串之内的换行符。
- 模式修饰符 **U** 交换元字符的匹配优先含义：「*」和「*?」交换，「+」和「+?」交换，等等。我猜这个模式修饰符的主要作用在于制造混乱，所以我完全不推荐使用它。

“Unknown Modifier”错误

有时候，手头程序忽然会报告“Unknown Modifier”错误。我绞尽脑汁希望找到问题所在，最终恍然大悟，原来自己在创建模式参数时忘了添加分隔符。

例如，我可能希望这样匹配 HTML tag：

```
preg_match('<(\w+)([^>]*)>', $html)
```

我的本意是，‘<’是正则表达式的一部分，但 `preg_match` 认为它是起始分隔符（我自己忘了设定分隔符，这还能怪谁呢？）。所以，这个参数被解释为‘`<(\w+)([^>]*)>`’，其中的正则表达式以灰色标注，模式修饰符以下画线标注。

在正则表达式中，「`(\w+)([^`」是不合法的，但是在发现并报告错误之前，正则引擎会试图将‘`]*)>`’解释为一串模式修饰符。但它们全都不是合法的模式修饰符，所以，当然会报告错误。

```
Warning: Unknown modifier ']'
```

显然，我需要使用分隔符：

```
preg_match('/<(\w+)(.*?)>/', $html)
```

除非我知道这里的 modifier 指的是 PHP 的 *pattern* 修饰符，否则此错误报告中给出的修饰符并不能让人明白，所以有时候我得花点时间才能找到问题所在。每次遇到这样的问题，我都觉得自己很傻，但幸运的是，没人知道我会犯这种低级的错误。

幸好，PHP5 最近版本的报错信息改成了这样：

```
Warning: preg_match(): Unknown modifier ']'
```

因为出现了函数名，我立刻就能反应过来。不过，有时候仍然需要花很多时间来查找漏写分隔符的问题，因为**不是每次都会报错**。比如下面这段程序：

```
preg_match('<(\w+)(.*?)>', $html)
```

尽管我忘了写分隔符，但「`(\w+)(.*?)`」仍然是合法的正则表达式。唯一的毛病在于它不能匹配我期望的结果。这种错误不易察觉，非常棘手。

Preg 函数罗列

The Preg Functions

本节从最基本的“这个正则表达式能否在字符串中找到匹配”开始，详细介绍各个函数。

preg_match

使用方法

```
preg_match(pattern, subject[, matches[, flags [, offset]]])
```

参数简介

pattern 分隔符包围起来的正则表达式，可能出现修饰符（☞444）。

subject 需要搜索的目标字符串。

matches 非强制出现，用来接受匹配数据。

flags 非强制出现，此标志位会影响整个函数的行为。这里只容许出现一个标志位，PREG_OFFSET_CAPTURE（☞452）。

offset 非强制出现，从 0 开始，表示匹配尝试开始的位置（☞453）。

返回值

如果找到匹配，就返回 true，否则返回 false。

讲解

最简单的用法是：

```
preg_match($pattern, $subject)
```

如果$pattern 在$subject 中能找到匹配，就会返回 true。下面有几个简单的例子：

```
if (preg_match('/\.(jpe?g|png|gif|bmp)$/i', $url)) {
    /* 图片的 URL */
}
-----------------------------------
if (preg_match('{^https?://}', $uri)) {
    /* URI 是 http 或 https */
}
-----------------------------------
if (preg_match('/\b MSIE \b/x', $_SERVER['HTTP_USER_AGENT'])) {
  /* 浏览器是 IE */
}
```

捕获匹配数据

preg_match 的第 3 个参数如果出现，则会用来保存匹配结果的信息。用户可以照自己的意愿使用任何变量，不过最常用的名字是$matches。在本书中，如果我在特定的例子之外提到$matches，指的就是“preg_match 接收的第 3 个参数”。

匹配成功之后，preg_match 返回 true，并按如下规则设置$matches：

```
$matches[0]是正则表达式匹配的所有文本
$matches[1]是第 1 组捕获型括号捕获的文本
$matches[2]是第 2 组捕获型括号捕获的文本
    ......
```

如果使用了命名分组，$matches 中也会保存对应的元素（下一节有这样的例子）。

第 5 章中（☞191）曾出现过这个简单的例子：

```
/* 输入完整路径，分离出文件名 */
if (preg_match('{ / ([^/]+) $}x', $WholePath, $matches))
  $FileName = $matches[1];
```

最好是在 preg_match 返回 true 的情况下用$matches(随便你怎么命名)。如果匹配不成功，会返回 false，或者错误（例如模式错误或函数标志位设置错误）。有的错误发生之后，$matches 是空数组，但也有时候它的值不会变化，所以我们不能认为，$matches 不为空就表示匹配功。

下面这个例子使用了 3 组捕获型括号：

```
/* 从 URL 中提取协议、主机名和端口号 */
if (preg_match('{^(https?):// ([^/:]+) (?: :(\d+) )? }x', $url, $matches))
{
   $proto = $matches[1];
   $host  = $matches[2];
   $port  = $matches[3] ? $matches[3] : ($proto == "http" ? 80 :443);
   print "Protocol: $proto\n";
   print "Host : $host\n";
   print "Port : $port\n";
}
```

数组结尾“未参与匹配”的元素会被忽略

如果一组捕获型括号没有参与最终匹配，它会在对应的$matches 中生成一个空字符串（注2）。需要说明的是，$matches 末尾的空字符串都会被忽略。在前面那段程序中，如果「(\d+)」参与了匹配，$matches[3]会保存一个数值，否则，$matches[3]根本就不会存在。

注 2：如果希望用 NULL 取代空字符串，请参考第 454 页的补充内容。

命名捕获

如果我们用命名捕获（☞138）重写之前的例子，正则表达式会长一些，不过代码更容易阅读：

```
/* 从 URL 中提取协议、主机名和端口号 */
if (preg_match('{^(?P<proto> https? ) ://
                  (?P<host> [^/:]+ )
             (?: : (?P<port> \d+ ) )? }x', $url, $matches))
{
   $proto  = $matches['proto'];
   $host   = $matches['host'];
   $port   = $matches['port'] ? $matches['port'] : ($proto== "http" ?80 : 443);
   print   "Protocol: $proto\n";
   print   "Host     : $host\n";
   print   "Port     : $port\n";
}
```

命名捕获看起来更清晰，这样我们不需要把`$matches`的内容复制给各个变量，就能直接使用变量名，而不是`$matches`，例如这样：

```
/* 从 URL 中提取协议、主机名和端口号 */
if (preg_match('{^(?P<proto> https? )://
                  (?P<host> [^/:]+ )
             (?: : (?P<port> \d+ ) )? }x', $url, $UrlInfo))
{
  if (! $UrlInfo['port'])
     $UrlInfo['port'] = ($UrlInfo['proto'] == "http" ? 80 : 443);
  echo "Protocol: ", $UrlInfo['proto'], "\n";
  echo "Host     : ", $UrlInfo['host'],  "\n";
  echo "Port     : ", $UrlInfo['port'],  "\n";
}
```

如果使用了命名捕获，按数字编号的捕获仍然会插入`$matches`。例如，在匹配`$url`（值为‘`http://regex.info`’）之后，之前例子中的`$UrlInfo`包含：

```
array
(
    0          =>  'http://regex.info',
    'proto'    =>  'http',
    1          =>  'http',
    'host'     =>  'regex.info',
    2          =>  'regex.info'
)
```

这样的重复有点浪费，但这是获得命名捕获的便捷和清晰所必须付出的代价。为清晰起见，我不推荐同时使用命名和数字编号来访问`$matches`的元素，当然用`$matches[0]`表示全局匹配例外。

请注意，数组中不包括编号为 3 和名称为‘`port`’的入口（entry），因为这一组捕获型括号没有参与到最终匹配中，而且处于最后（因此会被忽略☞450）。

还要提一点，尽管现在使用例如「(?P<2>…)」之类的数字命名并不会出错，但这种做法并不可取。PHP4 和 PHP5 在处理非正常情况时会有所区别，可能不会按照个人的意愿发展，所以最好还是不要使用数字来命名捕获分组。

更多的匹配细节：PREG_OFFSET_CAPTURE

如果设置了 preg_match 的第 4 个参数 *flags*，而且包含 PREG_OFFSET_CAPTURE（这也是 preg_match 目前能够接受的唯一标志位），则$matches 的每个元素不再是普通字符串，而是由两个元素构成的子数组，其中第 1 个元素是匹配的文本，第 2 个元素是这段文本在目标字符串中的偏移值（如果没有参与匹配，则为-1）。

偏移值从 0 开始，表示这段文本相对目标字符串的偏移值，即使设置了第 5 个参数$offset，偏移值的计算也不会变化。它们通常按照**字节**来计数，即使使用了模式修饰符 u 也是如此（☞447）。

来看个从 tag 中提取 HREF 属性的例子。HTML 的属性值两边可能是双引号、单引号，或者干脆没有引号，这样的值在下面这个正则表达式的第 1 组、第 2 组和第 3 组捕获型括号中被捕获：

```
preg_match('/href \s*=\s*(?: "([^"]*)" |\' ([^\']*)\v |([^\s\'">]+) )/ix',
        $tag,
        $matches,
        RPEG_OFFSET_CAPTURE);
```

如果$tag 包含：

```
<a name=bloglink href='http://regex.info/blog/' rel="nofollow">
```

匹配成功之后，$matches 的内容是：

```
array
(
    /* 全局匹配的数据 */
    0 => array ( 0 => "href='http://regex.info/blog/'",
                 1 => 17 ),
    /* 第 1 组括号的匹配数据 */
    1 => array ( 0 => "",
                 1 => -1 ),
    /* 第 2 组括号的匹配数据 */
    2 => array ( 0 => "http://regex.info/blog/",
                 1 => 23 )
)
```

$matches[0][0]包含正则表达式匹配的所有文本，$matches[0][1]表示匹配文本在目标字符串中的偏移值，按字节计数。

为了清晰起见，另一种获得$matches[0][0]的办法是：

```
substr($tag, $matches[0][1], strlen($matches[0][0]));
```

$matches[1][1]是-1，表示第 1 组捕获括号没有参与匹配。第 3 组也没有参与，但是因为之前提到的理由（☞450），结尾未参与匹配的捕获括号匹配的文本不会包含在$matches 中。

offset 参数

如果 preg_match 中设置了 *offset* 参数，引擎会从目标字符串的对应位置开始（如果 *offset* 是负数，则从字符串的末尾开始倒数）。默认情况下，offset 是 0（也就是说，从目标字符串的开头开始）。

请注意，offset 是按字节计数的，即使使用了模式修饰符 u 也是这样。如果设置不正确（例如从某个多字节字符的“内部”开始）会导致匹配失败。

即使 *offset* 不等于 0，PHP 也不会把这个位置标记为「^」——字符串的起始位置，它只表示正则引擎开始尝试的位置。不过，逆序环视倒是可以检查 *offset* 左边的文本。

preg_match_all

使用方法

```
preg_match_all(pattern, subject, matches [, flags [, offset ]])
```

参数简介

pattern 分隔符包围起来的正则表达式，可能出现修饰符（☞444）。

subject 需要检索的目标字符串。

matches 用来保存匹配数据的变量（**必须出现**）。

flags 非强制出现，标志位设定整个函数的功能：

PREG_OFFSET_CAPTURE （☞456）

和/或任意：

PREG_PATTERN_ORDER （☞455）
PREG_SET_ORDER （☞456）

offset 非强制出现，从 0 开始，表示目标字符串中匹配尝试开始的位置（与 preg_match 的 *offset* 参数相等☞453）。

返回值

preg_match_all 返回匹配的次数。

不包含任何内容的匹配，还是无法匹配

preg_match 返回的$matches 中，空字符串表示对应的括号没有参与匹配（当然，数组末尾的空字符串会被忽略）。因为匹配结果也可能是空字符串，我希望没有参与匹配的括号捕获的文本是 NULL。

所以，我自己编写了一个preg_match（我称其为 reg_match），首先使用 PREG_OFFSET_CAPTURE 标志位来获得所有括号内的自表达式匹配结果的详细信息，然后根据这些信息在$matches 中将对应的值设为 NULL：

```
function reg_match($regex, $subject, &$matches, $offset = 0)
{
    $result = preg_match($regex, $subject, $matches,
                         PREG_OFFSET_CAPTURE, $offset);
    if ($result) {
        $f = create_function('&$X', '$X = $X[1] < 0 ? NULL : $X[0]; ');
        array_walk($matches, $f);
    }
    return $result;
}
```

reg_match 的结果等于在不指定任何标志位的情况下调用 preg_match，区别在于，如果**某组括号没有参与匹配**，在 preg_match 中对应的元素为空字符串，而在 reg_match 中变成了 NULL。

讲解

preg_match_all 类似于 preg_match，只是在找到第一个匹配之后，它会继续搜索字符串，找到其他的匹配。每个匹配都会创建一个包含匹配数据的数组，所以最后 matches 变量就是一个二维数组，其中的每个子数组对应一次匹配。

这里有个简单的例子：

```
if (preg_match_all('/<title>/i', $html, $all_matches) > 1)
  print "whoa, document has more than one <title>!\n";
```

preg_match_all 要求必须出现第 3 个参数（也就是用来收集所有成功的匹配信息的变量）。所以，这个例子中虽然没有用到$all_matches，但仍然必须设置这个变量。

收集匹配数据

preg_match 和 preg_match_all 的另一个主要区别是第 3 个参数中的数据。preg_match 进行至多一次匹配，所以它把匹配的数据存储在 matches 变量中。与此不同的是，preg_

match_all 能匹配许多次，所以它的第 3 个参数保存了多个单次匹配的 matches。为了说明这种区别，我使用$all_matches 作为 preg_match_all 的变量名，而不是$preg_match 中常用的$matches。

preg_match_all 可以以两种方式在$all_matches 中存放数据，根据下面两个互斥的第 4 个参数 *flag*：PREG_PATTERN_ORDER 或是 PREG_SET_ORDER 来决定。

默认的排列方式是 PREG_PATTERN_ORDER 下面有个例子（我称其为“按分组编号的(collated)”——稍后将介绍）。如果没有设置标志位，这就是默认的配列方式：

```
$subject = "
Jack A. Smith
Mary B. Miller";
/* 不设置 flags 就采用 PREG_PATTERN_ORDER */
preg_match_all ('/^(\w+) (\w\.) (\w+)$/m', $subject, $all_matches);
```

$all_matches 的结果为：

```
array
 (
   /* $all_matches[0] 对应所有的全局匹配 */
   0 => array ( 0 => "Jack A. Smith", /* 第 1 次匹配的全部文本   */
                1 => "Mary B. Miller" /* 第 2 次匹配的全部文本   */ ),

   /* $all_matches[1] 对应第 1 组捕获型括号匹配的信息 */
   1 => array ( 0 => "Jack",  /* 第 1 次匹配中的第 1 组捕获型括号   */
                1 => "Mary"   /*第 2 次匹配的第 1 组捕获型括号     */ ),

   /* $all_matches[2] 对应第 2 组捕获型括号匹配的信息*/
   2 => array ( 0 => "A.",    /*第 1 次匹配中的第 2 组捕获型括号   */
                1 => "B."     /*第 2 次匹配中的第 2 组捕获型括号   */ ),

   /* $all_matches[3] 对应第 3 组捕获型括号匹配的信息*/
   3 => array ( 0 => "Smith", /*第 1 次匹配中的第 3 组捕获型括号   */
                1 => "Miller" /*第 2 次匹配中的第 3 组捕获型括号   */ )
 )
```

一共匹配了两次，每次都包含一个“全局匹配”字符串，以及 3 个捕获型括号对应的子字符串。我称其为“按分组编号的(collated)”，因为所有的全局匹配都存放在一个数组里（在$all_matches[0]，每次匹配中，第 1 组括号配的文本存放在另一个数组$all_matches[1] 中，依次类推。

默认情况下，$all_matches 是按分组编号的，但我们可设置 PREG_SET_ORDER 来改变它。

PREG_SET_ORDER 排列方式 如果设定了 PREG_SET_ORDER 标志位，就会采用“堆叠(stacked)”的排列方式。它会把第 1 次匹配的所有数据保存在$all_matches[0] 中，第 2 次匹配的所有数据保存在$all_matches[1] 中，依次类推。这就是我们检索字符串的顺序，把每次成功匹配的$matches 放进$all_matches 数组中。

下面是之前那个例子的 PREG_SET_ORDER 的版本：

```
$subject = "
Jack A. Smith
Mary B. Miller";

preg_match_all('/^(\w+) (\w\.) (\w+)$/m', $subject, $all_matches, PREG_SET_ORDER);
```

结果是：

```
array
 (
   /* $all_matches[0] 等价于 preg_match 的$matches */
   0 => array (0 => "Jack A. Smith",  /* 第1次整体匹配 */
                1 => "Jack",          /* 第1次整体匹配的第1个捕获型括号 */
                2 => "A.",            /* 第1次整体匹配的第2个捕获型括号 */
                3 => "Smith"          /* 第1次整体匹配的第3个捕获型括号 */ ),
   /* $all_matches[1] 等价于 preg_match 的$matches */
   1 => array (0 => "Mary B. Miller", /* 第1次整体匹配 */
                1 => "Mary",          /* 第2次整体匹配的第1个捕获型括号*/
                2 => "B.",            /* 第2次整体匹配的第2个捕获型括号*/
                3 => "Miller"         /* 第2次整体匹配的第3个捕获型括号*/ ),
 )
```

两种排列方式的总结如下：

类　型	标志位	说明及示例
按分组编号	PREG_PATTERN_ORDER	将各次匹配中同样编号的分组编在一起 $all_matches[$paren_num][**$match_num**]
堆叠	PREG_SET_ORDER	将每次匹配的数据集中保存 $all_matches[**$match_num**][$paren_num]

preg_match_all 和 PREG_OFFSET_CAPTURE 标志位

就像 preg_match 一样，也可以在 preg_match_all 中使用 PREG_OFFSET_CAPTURE，让$all_matches 的每个末端元素（leaf element）成为一个两个元素的数组（匹配的文本，以及按字节计算的偏移值）。也就是说，$all_matches 成为一个数组的数组的数组，这可真饶舌。如果你希望同时使用 PREG_OFFSET_CAPTURE 和 PREG_SET_ORDER，请使用逻辑运算符“or”来连接：

```
preg_match_all($pattern, $subject, $all_matches,
               PREG_OFFSET_CAPTURE | PREG_SET_ORDER);
```

preg_match_all 与命名分组

如果使用了命名分组，$all_matches 将会多出命名元素（同 preg_match 一样☞451）。这段程序：

```
$subject = "
Jack A. Smith
Mary B. Miller";
/*不设置 flags 就采用 PREG_PATTERN_ORDER */
preg_match_all('/^(?P<Given>\w+) (?P<Middle>\w\.) (?P<Family>\w+)$/m',
               $subject, $all_matches);
```

$all_matches 的结果是：

```
array
(   0        => array ( 0 => "Jack A. Smith", 1 => "Mary B. Miller" ),
    "Given"  => array ( 0 => "Jack",  1 => "Mary"   ),
    1        => array ( 0 => "Jack",  1 => "Mary"   ),
    "Middle" => array ( 0 => "A.",    1 => "B."     ),
    2        => array ( 0 => "A.",    1 => "B."     ),
    "Family" => array ( 0 => "Smith", 1 => "Miller" ),
    3        => array ( 0 => "Smith", 1 => "Miller" )
)
```

如果使用 PREG_SET_ORDER：

```
$subject = "
Jack A. Smith
Mary B. Miller";
preg_match_all('/^(?P<Given>\w+) (?P<Middle>\w\.) (?P<Family>\w+)$/m',
               $subject, $all_matches, PREG_SET_ORDER);
```

结果就是：

```
    array
    (
        0 => array ( 0      => "Jack A. Smith",
                     Given  => "Jack",
                     1      => "Jack",
                     Middle => "A.",
                     2      => "A.",
                     Family => "Smith",
                     3      => "Smith" ),
        1 => array ( 0      => "Mary B. Miller",
                     Given  => "Mary",
                     1      => "Mary",
                     Middle => "B.",
                     2      => "B.",
                     Family => "Miller",
                     3      => "Miller" )
)
```

我个人认为，在使用了命名分组之后，就应该去掉数字编号，因为这样程序更清晰，效率更高，不过，如果它们被保留了，你可以当它们不存在。

preg_replace

使用方法

```
preg_replace(pattern, replacement, subject [, limit [, count ]])
```

参数简介

pattern　分隔符包围起来的正则表达式，可能出现修饰符。*pattern* 也可能是一个 pattern-argument 字符串的数组。

replacement　replacement 字符串，如果 *pattern* 是一个数组，则 *replacement* 是包含多个字符串的数组。如果使用了模式修饰符 e，则字符串（或者是数组中的字符串）会被当作 PHP 代码（☞459）。

subject　需要搜索的目标字符串。也可能是字符串数组（按顺序依次处理）。

limit　非强制出现，是一个整数，表示替换发生的上限（☞460）。

count　非强制出现，用来保存实际进行的替换次数（只有 PHP5 提供，☞460）。

返回值

如果 *subject* 是单个字符串，则返回值也是一个字符串（*subject* 的副本，可能经过修改）。如果 *subject* 是字符串数组，返回值也是数组（包含 *subject* 的副本，可能经过修改）。

讲解

PHP 提供了许多对文本进行查找-替换的办法。如果查找部分可以用普通的字符串描述，`str_replace` 或者 `str_ireplace` 就更合适，但是如果查找比较复杂，就应该使用 `preg_replace`。

来看一个简单的例子：在 Web 开发中经常会遇到这样的任务，把信用卡号或电话号码输入一张表单。你是否经常看到“不要输入空格和连字符”的提示？要求用户按规则输入数据，还是由程序员做一点小小的改进，让用户可以照自己的习惯输入数据？哪种办法更好（注 3）？毕竟，这里我们的要求就是“清理”这样的输入数据：

```
$card_number = preg_replace('/\D+/', ' ', $card_number);
/* $card_number 只包含数字，或者为空 */
```

其中用 `preg_replace` 来去掉非数字字符。更确切的说，它用 `preg_replace` 来生成 `$card_number` 的副本，将其中的非数字字符替换为空（空字符串），把这个经过修改的副本赋值给`$card_number`。

注 3：显然，Web 开发中懒惰的程序员随处可见，所以我兄弟所做的“不要输入连字符和空格”的纪念堂很不幸地证明了这一点。参考 *http://www.unixwiz.net/ndos-shame.html*。

单字符串，单替换规则的 preg_replace

前面三个元素（*pattern*，*replacement* 和 *subject*）都是既可以为字符串，也可以为字符串数组。通常这三者都是普通的字符串，`preg_replace` 首先生成 subject 的副本，在其中找到 pattern 的第 1 次匹配，将匹配的文本替换为 replacement，然后重复这一过程，直到搜索到字符串的末尾。

在 replacement 字符串中，'`$0`' 表示匹配的所有文本，'`$1`' 表示第 1 组捕获型括号匹配的文本，'`$2`' 表示第 2 组，依次类推。请注意，美元符加数字的字符序列并不会引用变量，虽然它们在其他某些语言中有这种功能，但是 `preg_replace` 能识别简单的序列，并进行特殊处理。你可以使用一对花括号来包围数字，比如 '`${0}`' 和 '`${1}`'，这样就不会引起混淆。

这个简单的例子把 HTML 的 bold tag 转换为全部大写：

```
$html = preg_replace('/\b[A-Z]{2,}\b/', '<b>$0</b>', $html);
```

如果使用了模式修饰符 **e**(它只能出现在 `preg_replace` 中),replacement 字符串会作为 PHP 代码，每次匹配时执行，结果作为 replacement 字符串。下面这个扩展的例子把 bold tag 里的单词变为小写：

```
$html = preg_replace('/\b[A-Z]{2,}\b/e',
                     'strtolower("<b>$0</b>")',
                     $html);
```

如果正则表达式匹配的单词是 '`HEY`'，replacement 字符串中的`$0` 会被替换为这个值。结果，replacement 字符串就成了 '`strtolower("<b>HEY</b>")`'，执行这段 PHP 代码，结果就是 '`<b>hey</b>`'。

如果使用模式修饰符 **e**，replacement 字符串中的捕获引用会按照特殊的规定来插值：插值中的引号（单引号或双引号）会转义。如果不这样处理，插入的数值中的引号会导致 PHP 代码出错。

如果使用模式修饰符 **e**，在 replacement 字符串中引用外部变量，最好是在 replacement 字符串文本中使用单引号，这样变量就不会进行错误的插值。

这个例子类似于PHP内建的htmlspecialchars()函数：

```
$replacement = array ( '&' => '&amp; ',
                       '<' => '&lt; ',
                       '>' => '&gt; ',
                       '"' => '&quot; ');

$new_subject = preg_replace('/[&<">]/eS', '$replacement["$0"]', $subject);
```

需要注意，这个例子中的*replacement*使用了单引号字符串来避免$replacement变量插值，直到将其作为PHP代码执行。如果使用双引号字符串，在传递给preg_replace**之前**，插值就会进行。

可以用模式修饰符S用来提高效率（☞478）。

preg_replace的第4个参数用来设定替换操作次数的**上限**（单位是单个字符串-单个正则表达式，参见下一节）。默认值是-1，表示"没有限制"。

如果设置了第5个参数*count*（PHP4没有提供），它会用来保存preg_replace的实际替换的次数。如果你希望知道是否发生了替换，可以比较原来的目标字符串和结果，不过检查*count*参数效率更高。

多字符串，多替换规则

前一节已经提到，目标字符串通常是普通字符串，至少我们目前看到的所有例子都是如此。不过，*subject*也可以是一个字符串数组，这样搜索和替换是对每个字符串依次进行的。返回值也是由每个字符串经过搜索和替换之后的数组。

无论使用的是字符串还是字符串数组，*pattern*和*replacement*参数也可以是字符串数组，下面是各种组合及其意义：

Pattern	Replacement	行为
字符串	字符串	应用pattern，将每次匹配的文本替换为replacement
数组	字符串	轮流应用pattern，将每次匹配的文本替换为replacement
字符串	数组	轮流应用pattern，将每次匹配的文本替换为对应的replacement
数组	数组	不容许

如果subject参数是数组，则依次处理数组中的每个元素，返回值也是字符串数组。

请注意 *limit* 参数是以单个 pattern 和单个 subject 为单位的。它不是对所有的 pattern 和 subject 生效。返回的$count 则是所有 pattern 和 subject 字符串所进行操作次数的总合。

这里有一个 preg_replace 的例子，其中 pattern 和 replacement 都是数组。其结果类似于 PHP 内建的 htmlspecialchars() 函数，它保证处理过的文本符合 HTML 规范：

```
$cooked = preg_replace(
   /* 要匹配的文本...        */ array('/&/', '/</', '/>/', '/"/' ),
   /* 要替换的文本...        */ array('&amp;', '&lt;', '&gt;', '&quot;'),
   /* ...要操作的目标字符串 */ $text
);
```

如果输入的文本是：

```
AT&T --> "baby Bells"
```

$cooked 的值就是：

```
AT&amp;T --&gt; &quot;baby Bells&quot;
```

当然也可以预先准备好这些数组，下面的程序运行结果相同：

```
$patterns     = array('/&/', '/</', '/>/', '/"/' );
$replacements = array('&amp;', '&lt;', '&gt;', '&quot;');

$cooked = preg_replace($patterns, $replacements, $text);
```

preg_replace 能够接收数组作为参数是很方便的（这样程序员就不需要使用循环在各个 pattern 和 subject 中进行迭代），但是它的功能并没有增强。比如，各个 pattern 并不是“并行”处理的。但是，相比自己写 PHP 循环代码，内建的处理效率更高，而且更容易阅读。

为了说清楚，请参考这个例子，其中所有的参数都是数组：

```
$result_array = preg_replace($regex_array, $replace_array, $subject_array);
```

它等价于：

```
$result_array = array();
foreach ($subject_array as $subject)
{
   reset($regex_array);    // 准备遍历两个数组
   reset($replace_array); // 把数组指针恢复到开头位置

   while (list(,$regex) = each($regex_array))
   {
       list(,$replacement) = each($replace_array);
       // regex 和 replacemnet 已经准备就绪，应用到 subject ...
       $subject = preg_replace($regex, $replacement, $subject);
   }
   // 已经处理完所有的 regex，此 subject 处理完毕...
   $result_array[] = $subject; // ...附加到结果数组中
}
```

数组参数的排序问题 如果 *pattern* 和 *replacement* 都是字符串，它们会根据数组的内部顺序配对，这种顺序通常就是它们添加到数组中的先后顺序（*pattern* 数组中添加的第 1 个元素对应 *replacement* 数组中的第 1 个元素，依次类推）。也就是说，对于 array()创建的“文本数组”来说，排序没有问题，例如：

```
$subject = "this has 7 words and 31 letters";

$result = preg_replace(array('/[a-z]+/', '/\d+/'),
                       array('word<$0>', 'num<$0>'),
                       $subject);

print "result: $result\n";
```

「[a-z]+」对应‘word<$0>’，下面的「\d+」对应‘num<$0>’，结果就是

```
result: word<this> word<has> num<7> word<words> word<and> num<31> word<letters>
```

相反，如果 *pattern* 或 *replacement* 数组是多次填充的，数组的内部顺序可能就不同于 keys 的顺序（也就是说，由 keys 表示的数字顺序）。所以前一页的程序使用数组模拟 preg_replacement 的程序要使用 each 来按照数组的内部顺序遍历整个数组，而不关心它们的 keys 如何。

如果 *pattern* 或 *replacement* 数组的内部顺序不同于你希望匹配的顺序，可以使用 ksort() 函数来确保每个数组的实际顺序和外表顺序是相同的。

如果 *pattern* 和 *replacement* 都是数组，而 pattern 中元素的数目多于 replacement 中的元素，则会在 replacement 数组中产生对应的空字符串，来进行配对。

pattern 数组中的元素顺序不同，结果可能大不相同，因为它们是按照数组中的顺序来处理的。如果把例子中的 *pattern* 数组的顺序颠倒过来（把 *replacement* 数组中的顺序也颠倒过来），结果是什么呢？也就是说，下面代码的结果是什么呢？

```
$subject = "this has 7 words and 31 letters";
$result = preg_replace(array('/\d+/', '/[a-z]+/'),
                       array('num<\0>', 'word<\0>'),
                       $subject);
print "result: $result\n";
```

◆ 请翻到下页查看答案。

preg_replace_callback

使用方法

```
preg_replace_callback(pattern, callback, subject [, limit [, count ]])
```

参数简介

pattern　分隔符包围起来的正则表达式，可能出现修饰符（☞444）。也可能是字符串数组。

callback　PHP 回调函数，每次匹配成功，就执行它，生成 replacement 字符串。

subject　需要搜索的目标字符串。也可能是字符串数组（依次处理）。

limit　非强制出现，设定替换操作的上限（☞460）。

count　非强制出现，用来保存实际发生替换的次数（只在 PHP 5.1.0 中提供）。

返回值

如果 *subject* 是字符串，返回值就是字符串（其实是 *subject* 的一个副本，可能经过了修改）。如果 *subject* 是字符串数组，返回值就是数组（每个元素都是 *subject* 中对应元素的副本，可能发生了修改）。

讲解

`preg_replace_callback` 类似于 `preg_replace`，只是 replacement 参数变成了 PHP 回调函数，而不是字符串或是字符串数组。它有点像使用模式修饰符 **e** 的 `preg_replace`（☞459），但是效率更高（如果 replacement 部分的代码很复杂，这种办法更易于阅读）。

请参考 PHP 文档获得更多关于回调的知识，不过简单地说，PHP 回调引用（以许多种方式中的一种）一个预先规定的函数，以预先规定的参数，返回预先规定的值。在 `preg_replace_callback` 中，每次成功匹配之后都会进行这种调用，参数是`$matches` 数组。函数的返回值用作 `preg_replace_callback` 作为 replacement。

回调可以以三种方式引用函数。一种是直接以字符串形式给出函数名；另一种是用 PHP 内建的 `create_function` 生成一个匿名函数。稍后我们会看到使用这两种方法的例子。第三种方式本书没有提及，它采用面向对象的方式，由一个包含两个元素（分别是类名和方法名）的数组构成。

测验答案

◆ 462 页问题的答案

462 页问题中的程序运行结果如下（为了适应排版，进行了折行）：

```
result: word<this> word<has> word<num><7> word<words>
word<and> word<num><31> word<letters>
```

如果这两处粗体内容出乎你的意料，原因在于，使用多个正则表达式的 preg_match（使用 *pattern* 数组）并不会“并行”处理这些 pattern，而是依次进行。

在这个例子中，第 1 组 *pattern/replacement* 会在 subject 中添加两个 num<…>，这两个‘num’会被数组中的下一个 *pattern* 匹配。然后每个‘num’变成‘word<num>’，最终得到这个意料之外的结果。

这个例子告诉我们，如果 preg_replace 使用了多个 *pattern*，一定要注意安排它们的顺序。

下面这个例子用 preg_replace_callback 和辅助函数重写了第 460 页的程序。callback 参数是一个字符串，包含辅助函数的名字：

```
$replacement = array ( '&' => '&amp;',
                       '<' => '&lt;',
                       '>' => '&gt;',
                       '"' => '&quot;');
/*
 * 匹配成功之后，$matches[0]中保存的是需要转换为 HTML 的字符串，以此为接受参数，返回
 * HTML 字符串。因为此函数只在确保安全的情况下调用，此处不考虑意外情况
 */
function text2html_callback($matches)
{
    global $replacement;
    return $replacement[$matches[0]];
}
$new_subject = preg_replace_callback('/[&<">]/S',    /* pattern  */
                                     "text2html_callback", /* callback */
                                     $subject);
```

如果$subject 的值是：

```
"AT&T" sounds like "ATNT"
```

则$new_subject 的值就是：

```
&quot;AT&amp;T&quot; sounds like &quot;ATNT&quot;
```

本例中的 text2html_callback 是普通的 PHP 函数，用作 preg_replace_callback 中的回调函数，它的接收参数是$matches 数组（当然，这个变量可以随意命名，不过我选择遵循之前使用$matches 的惯例）。

为完整起见，下面我给出使用匿名函数的办法（使用 PHP 内建的 `create_function` 函数）。这段程序产生的`$replacement` 变量与上面一样。函数体也相同，只是此时函数没有名字，只能在 `preg_replace_callback` 中使用：

```
$new_subject = preg_replace_callback('/[&<">]/S',
                  create_function('$matches',
                                  'global $replacement;
                                  return $replacement[$matches[0]];'),
                  $subject);
```

使用 callback，还是模式修饰符 e

如果处理不复杂，使用模式修饰符的程序比 `preg_replace_callback` 更容易看懂。但是，如果效率很重要，那么请记住，如果使用模式修饰符 **e**，每次匹配成功之后都需要检查作为 PHP 代码的 replacement 参数。相比之下，`preg_replace_callback` 的效率就要高许多（如果使用回调，PHP 代码只需要审查 1 次）。

preg_split

使用方法

```
preg_split(pattern, subject [, limit, [ flags ]])
```

参数简介

pattern 分隔符包围起来的正则表达式，可能还有修饰符(☞444)。

subject 需要分割的目标字符串。

limit 非强制出现，是一个整数，表示切分之后元素的上限。

flags 非强制出现，此标志位影响整个切割行为，以下三项可以随意组合：

PREG_SPLIT_NO_EMPTY
PREG_SPLIT_DELIM_CAPTURE
PREG_SPLIT_OFFSET_CAPTURE

它们的讲解从第 468 页开始。多个标志位使用二元运算符“或”来连接（与第 456 页一样）。

返回值

返回一个字符串数组。

讲解

`preg_split` 会把字符串的副本切分为多个片段，以数组的形式返回。非强制出现参数 *limit*

设定返回数组中元素数目的上限（如果需要，最后的元素包括“其他所有字符”）。可以设定不同的标志位来调整返回的方式和内容。

从某种意义上来说，preg_split 做的是与 preg_match_all 相反的事情：它找出目标字符串中不能由正则表达式匹配的部分。或者更传统地说，preg_split 返回的是，将目标字符串中正则表达式匹配的部分删去之后的部分。preg_split 大概相当于 PHP 中内建的简单 explode 函数，不过使用的是正则表达式，而且功能更强大。

来看个简单的例子，如果某家金融网站需要接收用空格分隔的股票行情。可以使用 explode 拆分这些行情数据：

```
$tickers = explode(' ', $input);
```

不过，如果输入数据时不小心输入了不只一个空格，这个程序就不能处理了。更好的办法是使用 preg_split，用正则表达式「\s+」来切分：

```
$tickers = preg_split('/\s+/', $input);
```

除了明确运用“用空格切分”的规则之外，用户也通常使用逗号（或者是逗号加空格）来分隔，比如‘YHOO, MSFT, GOOG’。这些情况也很容易处理：

```
$tickers = preg_split('/[\s,]+/', $input);
```

针对上面的数据，$tickers 得到的是包含 3 个元素的数组：‘YHOO’、‘MSFT’和‘GOOG’。

如果输入的数据是逗号分隔的（例如给照片标记 tag 时使用的“Web 2.0,”），就需要用「\s*,\s*」来处理：

```
$tags = preg_split('/\s*,\s*/', $input);
```

比较「\s*,\s*」和「[\s,]+」很能说明问题。前者用逗号来切分（逗号必须出现），但也会删去逗号两边的空白字符。如果输入‘123,,,456’，则能够进行 3 次匹配（每次匹配一个逗号），返回 4 个元素：‘123’，两个空字符串，最后是‘456’。

另一方面，「[\s,]+」会使用任何逗号、连续的逗号、空白字符，或者是空白字符和逗号的结合来切分。在‘123,,,456’中，它一次就能匹配 3 个逗号，返回两个元素，‘123’和‘456’。

limit 参数

limit 参数用来设定切分之后数组长度的上限。如果搜索尚未进行到字符串结尾时，切分的片段的数目已经达到 *limit*，则之后的内容会全部保存到最后的元素当中。

来看个例子，我们需要手工解析服务器返回的 HTTP response。按照标准，header 和 body 的分隔是四字符序列‘\r\n\r\n’，不幸的是，有的服务器使用的却是‘\n\n’。幸好，我们有 preg_split，很容易处理这两种情况。假设整个 response 保存在$response 中：

```
$parts = preg_split('/\r? \n \r? \n/x', $response, 2);
```

header 保存在$parts[0]中，而 body 保存在$parts[1]中（使用模式修饰符 S 是为了提高效率☞478）。

第 3 个参数，即 *limit* 的值等于 2，表示 subject 字符串最多只能切分成两个部分。如果找到了一个匹配，匹配之前的部分（也就是 header）会成为返回值的第一个元素。因为“字符串的其他部分”是第 2 个元素，这样就到达了上限，它（我们知道是 body）会原封不动地作为返回值中的第二个元素。

如果没有 *limit*（或者 *limit* 等于-1，这两种情况是等价的），preg_split 会尽可能多地切分 subject 字符串，这样 body 可能也会被切分为许多段。设置上限并不能保证返回的数组中包含的元素就等于这个数，而只是保证最多包含这么多元素（阅读关于 PREG_SPLIT_DELIM_CAPTURE 的小节，你会发现甚至这种说法也不完全对）。

在两种情况下，应该人为设置上限。我们已经见过一种情况：希望最后的元素包含“其他所有内容”。在前一个例子中，一旦第一段（header）被切分出来，我们就不希望再对其他部分（body）进行切分。所以，把上限设为 2 会保留 body。

如果用户知道自己不需要切割出来所有元素，也可以设定上限，提高效率。例如，如果$data 字符串包含以「\s*,\s*」分隔的许多字段（比如姓名、地址、年龄，等等），而只需要前面两个，就可以把 *limit* 设置为 3，这样 preg_split 在切分出前两个字段之后就不会继续工作：

```
$fields = preg_split('/ \s*, \s*/x', $data, 3);
```

这样其他内容都保存在最后的第 3 个元素中，我们可以用 array_pop 来删除，或者置之不理。

如果你希望在没有设置上限的情况下使用任何 preg_split 标志位（下一节讨论），则必须提供一个占位符，将 *limit* 设置为-1，它表示“没有限制”。相反，如果 *limit* 等于 1，则表示“不需要切分”，所以它并不常用。上限等于 0 或者-1 之外的任何负数都没有定义，所以请不要使用它们。

flag 参数

preg_split 中可以使用的3个标志位都会影响函数的功能。它们可以单独使用，也可以用二元运算符“or”连接（参见第456页的例子）。

PREG_SPLIT_OFFSET_CAPTURE 就像在 preg_match 和 preg_match_all 中使用 PREG_OFFSET_CAPTURE 一样，这个标志位会修改结果数组，把每个元素变为包含两个元素的数组（元素本身和它在字符串中的偏移值）。

PREG_SPLIT_NO_EMPTY 这个标志位告诉 preg_split 忽略空字符串，不把它们放在返回数组中，也不记入 *limit* 的统计。对目标字符串的起始位置、结尾位置，或是空行的匹配，都会带来空字符串。

下面来改进前面的“Web 2.0”的 tag 的例子（☞466），如果$input 为‘party,,fun’，那么：

```
$tags = preg_split('/\s*, \s*/x', $input);
```

得到的$tags 包含3个元素：‘party’、空字符串，然后是‘fun’。空字符串是逗号的两次匹配之间的“空白”。

如果设置了 PREG_SPLIT_NO_EMPTY 标志位：

```
$tags = preg_split('/\s*, \s*/x', $input, -1, PREG_SPLIT_NO_EMPTY);
```

结果数组只包含‘party’和‘fun’。

PREG_SPLIT_DELIM_CAPTURE 这个标志位在结果中包含匹配的文本，以及进行此次切分的正则表达式的捕获括号匹配的文本。来看个简单的例子，如果字符串中各个字段是以‘and’和‘or’来联系的，例如：

```
DLSR camera and Nikon D200 or Canon EOS 30D
```

如果不使用 PREG_SPLIT_DELIM_CAPTURE，

```
$parts = preg_split('/\s+ (and|or) \s+ /x', $input);
```

得到的$parts 是：

```
array ('DLSR camera', 'Nikon D200', 'Canon EOS 30D')
```

分隔符中的匹配内容被去掉了。不过，如果使用了 PRE_SPLIT_DELIM_CAPTURE 标志位（并且用-1 作为 *limit* 参数的占位符）：

```
$parts = preg_split('/\s+ (and|or) \s+ /x', $input, -1,
                    PREG_SPLIT_DELIM_CAPTURE);
```

$parts 包含了捕获型括号匹配的分隔符：

```
array ('DLSR camera', 'and', 'Nikon D200', 'or', 'Canon EOS 30D')
```

此时，每次切分会在结果数组中增加一个元素，因为正则表达式中只有一组捕获型括号。然后我们就能够遍历$parts中的元素，对找到的‘and’和‘or’进行特殊处理。

请注意，如果使用了非捕获型括号（如果 pattern 参数为‘/\s+(?:and|or)\s+/’），PREG_SPLIT_DELIM_CAPTURE 标志位不会产生任何效果，因为它只对捕获型括号有效。

来看另一个例子，第 466 页分析股市行情的例子：

```
$tickers = preg_split('/[\s,]+/', $input);
```

如果我们添加捕获型括号，以及 PREG_SPLIT_DELIM_CAPTURE

```
$tickers = preg_split('/([\s,]+)/', $input, -1, PREG_SPLIT_DELIM_CAPTURE);
```

结果$input 中的任何字符都没有被抛弃，它只是切分之后保存在$tickers 中。处理$tickers数组时，你知道编号为奇数的元素是「([\s,]+)」匹配的。这可能很有用，如果在向用户显示错误信息时，可以对不同的部分分别进行处理，然后将它们合并起来，还原出输入的字符串。

还有一点需要注意，通过 PREG_SPLIT_DELIM_CAPTURE 添加的元素不会影响切分上限。只有在这种情况下，结果数组中的元素数目才可能超过上限（如果正则表达式中的捕获型括号很多，则元素就要更多）。

结尾的未参与匹配的捕获型括号不会影响结果数组。也就是说，如果一组捕获型括号没有参与最终匹配（参见 450 页），可能会也可能不会在结果数组中添加空字符串。如果编号更靠后的捕获型括号参与了最终匹配，就会增加，否则就不会。请注意，如果使用了 PREG_SPLIT_NO_EMTPY 标志位，结果会有变化，因为空字符串肯定会被抛弃。

preg_grep

使用方法

```
preg_grep( pattern, input[, flags])
```

参数简介

pattern 分隔符包围起来的正则表达式，可能出现修饰符。

input 一个数组，如果它们的值能够匹配 pattern，则其值会复制到返回的数组中。

flags 非强制出现，此标志位 PREG_GREP_INVERT 或者是 0。

返回值

一个数组，包含 *input* 中能够由 *pattern* 匹配的元素（如果使用了 PREG_GREP_INVERT 标志位，则包括不能匹配的元素）。

讲解

preg_grep 用来生成 *input* 数组的副本，其中只保留了 value 能够匹配（如果使用了 PREG_GREP_INVERT 标志位，则不能匹配）*pattern* 的元素。此 value 对应的 key 会保留。

来看个简单的例子

```
preg_grep('/\s/', $input);
```

它返回$input 数组中的，由空白字符构成的元素。相反的例子是：

```
preg_grep('/\s/', $input, PREG_GREP_INVERT);
```

它返回不包含空白字符的元素。请注意，第二个例子不同于：

```
preg_grep('/^\S+$/', $input);
```

因为后者不包括空（长度为 0）值元素。

preg_quote

使用方法

```
preg_quote( input [, delimiter ])
```

参数简介

input 希望以文字方式用作 *pattern* 参数的字符串（☞444）。

delimiter 非强制出现的参数，包含 1 个字符的字符串，表示希望用在 *pattern* 参数中的分隔符。

返回值

preg_quote 返回一个字符串，它是 *input* 的副本，其中的正则表达式元字符进行了转义。如果指定了分隔符，则分隔符本身也会被转义。

讲解

如果要在正则表达式中以文字方式使用某个字符串，可以用内建的 preg_quote 函数来转义其中可能产生的正则表达式元字符。如果指定了创建 pattern 时使用的分隔符，字符串中的分隔符也会被转义。

preg_quote 是专门应对特殊情况的函数，在许多情况下没有用，不过这里有个例子：

```
/* 输入$MailSubject，判断$MailMessage 对应主题 */
$pattern = '/^Subject:\s+(Re:\s*)*' . preg_quote($MailSubject, '/') . '/mi';
```

如果$MailSubject 包含下面的字符串

```
**Super Deal** (Act Now!)
```

最后$pattern 就会是

```
/^Subject:\s+(Re:\s*)*\*\*Super Deal\*\* \(Act Now\!\)/mi
```

这样就可以作为 preg 函数的 pattern 参数了。

如果指定的分隔符是'{'之类对称的字符，那么对应的字符（例如'}'）不会转义，所以请务必使用非对称的分隔符。

同样，空白字符和'#'也不会转义，所以结果可能不适于用 x 修饰符。

这样说来，在把任意文本转换为 PHP 正则表达式的问题上，preg_quote 并不是完善的解决办法。它只解决了"文本到正则表达式"的问题，而没有解决"正则表达式到 pattern 参数"的问题，任何 preg 函数都需要这一步。下一节给出了解决办法。

"缺失"的 preg 函数

"Missing" Preg Functions

PHP 内建的 preg 函数已经提供了繁多的功能，但是有时候我仍然发现它们不够用。一个例子是我自己开发的 preg_match（☞454）。

我发现，另一类需要提供自己的支持函数的情形是，正则表达式不是在程序内部通过 pattern 参数字符串提供的，而是来自程序外部（例如，从文件读入，或者是在 Web 表单中提交）。下一节我们将会看到，把纯粹的正则表达式字符串转换为适合 patern 参数使用的形式也很复杂。

同样，在使用这些正则表达式之前，通常都必须验证他们的语法正确性。我同样会讲解这些问题。

与本书中的所有程序代码一样，下一页的函数也可以从我的网站下载：*http://regex.info/*。

preg_regex_to_pattern

如果正则表达式包含在字符串中（可能是从配置文件读入，或者通过 Web 表单提交），在 preg 函数中使用时，首先必须在两端加上分隔符，才能生成一个 preg 函数能用的 pattern 参数。

问题所在

许多时候，把正则表达式转换为 pattern 参数只不过是在两端加上斜线而已。这样，正则表达式字符串 `'[a-z]+'` 就成了 `'/[a-z]+/'`，可以用作 preg 函数的 pattern 参数的字符串。

如果正则表达式中包含用作分隔符的字符，情况就很复杂。例如，正则表达式是 `'^http://([^/:]+)'`，仅仅在两端添加反斜线得到 `'/^http://([^/:]+)/'`，用作 pattern 时，结果会是"Unknown modifier /"。

第 448 页已经介绍过，这个错误信息是因为字符串中的前两个斜线字符被当作分隔符，之后的部分（在这里就是第 3 个斜线之后的部分）被当作修饰符序列了。

解决之道

有两种办法能解决内嵌分隔符的问题。之一是选择正则表达式中没有出现的分隔符，如果需要手工构造 pattern-modifier 字符串，那么这当然是推荐的办法了。所以我在第 444、449 和 450 页（还有许多）的例子中使用`{…}`作为分隔符。

要选出正则表达式中没有出现的分隔符可能并不容易（甚至不可能），因为字符串中可能包含所有分隔符，或者你不能预先知道需要处理的文本。在实际应用正则表达式时这需要特别关注，所以最简单的办法是使用第二种：选择一个分隔符，然后对正则表达式字符串中出现的此分隔符进行转义。

问题可能比初看起来要困难许多，因为你必须关注某些重要的细节。例如，在目标字符串末尾的转义必须进行特殊处理，保证它不会转义紧跟在后面的分隔符。

下面的函数接收正则表达式字符串，以及可能出现的 pattern-modifier 字符串，返回一个可以用于 preg 函数的 pattern 字符串。代码中难看的反斜线（正则表达式和 PHP 子串转义）或

许是你见过的最复杂的表示；这段代码并不容易读懂（如果你希望补习 PHP 单引号字符串的语意，请参考第 444 页）。

```
/*
 * 输入字符串形式的正则表达式（或许是 pattern-modifier 字符串），返回适合 preg 函数
 * 的字符串，此表达式包含在分隔符之内，后面可能还跟有修饰符
 */
function preg_regex_to_pattern($raw_regex, $modifiers = "")
{
  /*
   * 进行转换需要在 pattern 两端添加分隔符（这里使用斜线）并添加修饰符
   * 必须转义表达式内部的分隔符，表达式结尾的转义必须特殊处理，否则它会转义最后的分隔符
   * 不能盲目转义表达式内部的分隔符，因为表达式内部可能包括已经转义的分隔符
   * 例如，如果表达式是'\/'，盲目转义得到'\\/'，最终结果是'/\\//'，这显然不对
   *
   * 应当把表达式分为三类：已转义的字符、未转义的斜线（需要处理）和其他字符。
   * 还需要注意字符串结尾的转义
   */
if (! preg_match('{\\\\(|:/|$)}', $raw_regex)) /* 后面是'\'或EOS的'/'  */
{
    /* 不存在已转义的斜线，末尾也没有转义，直接转义其中的斜线即可 */
    $cooked = preg_replace('!/!', '\/', $raw_regex);
}
else
{
    /* 用来解析$raw_regex 的 pattern
     * 捕获型括号内的两个部分需要转义 */
    $pattern = '{ [^\\\\/]+ | \\\\. |( / |\\\\$ ) }sx';

    /* $raw-regex 中$pattern 的每次成功匹配都需要调用回调函数
     * 如果$matches[1]不为空，返回转义后的结果
     * 否则不做修改直接返回 */
    $f = create_function('$matches', '     /* 这个长长的
         if (empty($matches[1]))          . /* 单引号
             return $matches[0];            /* 字符串就是
         else                               /* 函数的
             return "\\\\" . $matches[1];  /* 代码
     ');
    /* 将$pattern 应用到$raw_regex，得到$cooked */
    $cooked = preg_replace_callback($pattern, $f, $raw_regex);
  }
  /* 现在可以在$cooked 两端添加分隔符了，然后附上修饰符，然后返回 */
  return "/$cooked/$modifiers";
}
```

每次需要的时候重新写这个函数太麻烦，于是我将它封装到一个函数中（我希望它会成为 preg 套件中的内建函数）。

有兴趣的读者不妨想想函数尾部 `preg_replace_callback` 使用的正则表达式：它如何工

作，以及回调函数如何遍历整个 pattern 字符串，转义每个未转义的斜线，而不修改已转义的斜线。

对未知的 Pattern 参数进行语法检查

Syntax-Checking an Unknown Pattern Argument

在正则表达式两端添加分隔符之后，我们确信它适于用作 preg 函数的 pattern 参数了，但是原来的正则表达式还没有经过语法正确性检验。

举例来说，如果原始的正则表达式是‘`*.txt`’，因为某些人希望使用文件群组功能（☞4）而不是正则表达式，那么 `preg_regex_to_pattern` 返回的就是`/*.txt/`。这个正则表达式当然不合法，所以程序会发出警报（如果启用了警报功能）：

```
Compilation failed: nothing to repeat at offset 0
```

PHP 没有内建检测 pattern 参数及其正则表达式的语意是否合法的函数，不过我为读者提供了一个。

`preg_pattern_error` 会对 pattern 参数进行简单测试，试图使用此正则表达式——在函数当中有一行调用 `preg_match`。函数的其他部分使用关注 PHP 的管理功能处理 `preg_match` 可能显示的错误信息。

```
/*
 * 如果输入的 pattern 或其中的正则表达式参数语法不正确，输出错误信息
 * 否则（语法正确）返回 false。
 */
function preg_pattern_error($pattern)
{
    /*
     * 要判断 pattern 是否有错，直接尝试使用它。
     * 检测和捕获此错误却不容易，尤其是希望获得友好提示，而且不修改全局状态
     * 所以如果打开了‘track_errors’，则保存$php_errormsg ，之后恢复它的状态
     * 如果没有打开，则打开它（因为需要用到），完成之后再关闭
     */
    if ($old_track = ini_get("track_errors"))
        $old_message = isset($php_errormsg) ? $php_errormsg : false;
    else
        ini_set('track_errors', 1);
    /* 现在确认 track_errors 已经打开 */

    unset($php_errormsg);
    @ preg_match($pattern, ""); /*尝试使用 pattern */
    $return_value = isset($php_errormsg) ? $php_errormsg : false;

    /* 现在捕获了需要的内容，恢复全局状态 */
    if ($old_track)
        $php_errormsg = isset($old_message) ? $old_message : false;
    else
        ini_set('track_errors', 0);

    return $return_value;
}
```

对未知正则表达式进行语法检查

Syntax-Checking an Unknown Regex

这个函数使用刚刚开发的功能来检查一个纯正则表达式（没有分隔符也没有模式修饰符）的语法。如果语法不正确，会返回对应的错误信息，如果语法正确，返回 false。

```
/*
 * 如果表达式语法错误，返回错误信息，如果语法正确返回 false
 */
function preg_regex_error($regex)
{
    return preg_pattern_error(preg_regex_to_pattern($regex));
}
```

递归的正则表达式

Recursive Expressions

preg 引擎所属的流派的大多数方面都在第 3 章中有所介绍，但是此流派还提供了某些新的有意思的功能用于匹配嵌套结构：递归的表达式。

序列「(?R)」表示“在此处递归应用整个表达式”，而「(?*num*)」表示“在此处递归应用 num 所对应编号的捕获型括号中的序列”。命名捕获的括号则使用「(?P>*name*)」表示法。

下面几节展示了常见的递归。递归在扩展的“tagged data”例子（☞481）中占有重要地位。

匹配嵌套括号内的文本

Matching Text with Nested Parentheses

基本的递归的例子是匹配嵌套的括号内的文本，下面是一种办法：「(?:[^()]++|\((?R)\))*」。

这个表达式匹配任意多个双多选分支结构。第 1 个多选分支「[^()]++」匹配除括号之外的任何字符。因为外面有「(?:…)*」，这个多选分支要求使用占有优先的加号，避免“无休止匹配”（☞226）。

另一个多选分支「\((?R)\)」才是问题的关键。它匹配一对括号，其中可以包括任何字符（只要括号的嵌套是格式正确的）。这里的“之间”的部分是整个正则表达式希望匹配的内容，也就是我们可以通过「(?R)」直接递归使用整个正则表达式的原因。

这个表达式本身可以正常工作，但如果要添加任何字符，请务必小心，因为调用「(?R)」时添加的任何字符同样会递归。

如果使用这个正则表达式来校验一个括号配对不正确的字符串，你可能希望在两端加上「^…$」来确保“整个字符串”。这是不对的，因为添加的行锚点会被递归应用到整个字符串之中，导致匹配失败。

递归引用一组捕获型括号

「(?R)」结构会递归引用整个正则表达式，但也可以使用「(?*num*)」结构引用到其中的子集。它递归引用编号为 *num*th 的捕获型括号内的子表达式（注 4）。如果用「(?*num*)」来思考，「(?0)」就等于「(?R)」。

我们可以使用这种部分递归因来解决前面那一节中出现的问题：在添加「^…$」之前，我们用一个捕获型括号把正则表达式的主体部分围起来，然后在以前使用「(?R)」的地方使用「(?1)」。添加捕获型括号是为了让「(?1)」能够引用，你或许还记得，这就是上一节匹配嵌套括号的表达式。「^…$」加在这些括号之外，这样我们就不会对它们进行递归调用：「^((?:[^()]++|\((**?1**)\))*)$」。

正则表达式中下画线的部分是第 1 组捕获型括号，所以每次遇到「(?1)」都会重新应用。

下面 PHP 代码中的正则表达式会报告$text 中的括号能否配对：

```
if (preg_match('/^ ( (?: [^()]++ | \( (?1) \) )* ) $/x', $text))
   echo "text is balanced\n";
else
   echo "text is unbalanced\n";
```

通过命名捕获进行递归引用

如果需要递归调用的自表达式处于命名捕获（☞138）中，就可以使用「(?P>*name*)」进行递归引用，而不是之前的「(?*num*)」表示法。使用这个表示法，我们的例子就成了：

「^(**?P<stuff>** (?: [^()]++|\((**?P>**stuff) \))*)$.」

注 4：严格地说，「(?*num*)」不算递归引用，这是因为，「(?*num*)」结构本身并不必然是编号为 *num*th 的捕获型括号中的子表达式的一部分。此时，引用可以被看作“子程序调用”。

这个表达式可能看起来很复杂，用模式修饰符 x 可以看得更清楚一点：

```
$pattern = '{
    # 正则表达式从此处开始...
    ^
        (?P<stuff>
        # 这一组括号内的所有内容都被命名为"stuff."
        (?:
        [^()]++                    # 除括号之外的任何字符
        |
        \( (?P>stuff) \)           # 开括号，更多"stuff,"然后是闭括号
    )*
)
$
# 正则表达式结束
}x'; # 'x'是模式修饰符
if (preg_match($pattern, $text))
    echo "text is balanced\n";
else
    echo "text is unbalanced\n";
```

关于占有优先量词的补充

我会对表达式中的占有优先量词做最后的补充。如果外部的「(?:…)*」是占有优先的，内部就不必使用「[^()]++」。为了阻止这个表达式进入无休止匹配，其中之一（或者是两者）必须是占有优先的。如果不能使用占有优先量词和固化分组（☞259），则需要去掉所有的量词「(?:[^()]|\((?R)\))*」。

这样会降低效率，但是至少不会进入无法终止的匹配。要提高效率，可以使用第 6 章介绍的“消除循环”的技巧，得到「[^()]*(?:\((?R)\)[^()]*)*」。

不能回溯到递归调用之内

No Backtracking Into Recursion

preg 正则流派的递归语意的重要特性之一是，它会把递归结构匹配的所有内容当作固化分组括号匹配的内容（☞259）。也就是说，递归结构不会部分“交还（unmatch）”某些已经匹配的内容来实现全局匹配（而是导致整个匹配失败）。

这里说的“部分”是很重要的，因为回溯时，递归调用匹配的所有文本可以作为一个整体来交还。但不容许回溯到递归调用之内的某个状态。

匹配一组嵌套的括号

Matching a Set of Nested Parentheses

上文已经介绍过如何匹配包含规范配对括号的行，这里我会介绍匹配对称括号的办法（其中可能包含更多的嵌套括号）：「`\((?:[^()]++|(?R))*\)`」。

这个表达式的组成部分与之前的一样，但是顺序安排略有不同。同样，如果你希望把它当作大的正则表达式的一部分，应该把它包括在捕获型括号中，并且修改「`(?R)`」来递归地引用对应的自表达式，例如「`(?1)`」（必须使用这组捕获型括号对应的编号）。

效率

PHP Efficiency Issues

PHP 的 preg 程序使用的 PCRE 是经过优化的 NFA 正则引擎，所以第 4 到 6 章介绍的许多技巧都可以直接应用。其中包括对关键部分进行性能测试，根据实际数据而不是从理论分析来比较程序的快慢。第 6 章给出了 PHP 的性能测试的例子（☞234）。

如果程序对时间要求很严格，请记住两点，回调函数通常要比模式修饰符 e 更快（☞465），在太长的目标字符串中使用命名捕获必须进行更多的数据拷贝。

程序运行中，遇到正则表达式会编译，但是 PHP 有一个容量高达 4 096 个正则表达式的大型缓存（☞242），所以实际上，特殊的 pattern 字符串只需要在第一次遇到的时候编译。

模式修饰符 S 值得单独介绍：它会“研究(study)”一个正则表达式，试图进行更快的匹配（它与 Perl 的 `study` 函数不相关，Perl 的 `study` 函数研究的是目标字符串，而不是正则表达式☞359）。

模式修饰符 S："研究"

The S Pattern Modifier: "Study"

使用模式修饰符 S 告诉正则引擎，在应用这个正则表达式之前，花一点时间（注 5）来研究，希望这些多花的时间是值得的。但是，也可能这样做之后也不会提升速度，但是某些情况下，速度的提升是与数据规模相关的。

现在有良好的标准来判断哪种情况下此功能具有价值：它是第 6 章所说的开头字符组识别优化（☞247）的增强。

注 5：确实只是一点时间。对于一般的 CPU，非常长非常复杂的正则表达式，使用模式修饰符 *S* 只需要多花百分之一到千分之一秒的时间。

首先要告诉你的是，除非你希望对大规模的文本应用某个正则表达式，否则不太可能节省多少时间。只有把同一个表达式应用到大规模的文本，或者大量小规模文本时，才需要考虑模式修饰符 **S**。

不使用模式修饰符 S，标准优化

来看个简单的例子「<(\w+)」。从这个表达式中我们可以看出，每次匹配必须以字符‘<’开头。正则引擎能够（preg 套件必然会这样做）利用这一点，预先在目标字符串中搜索‘<’，只在这些位置应用完整的正则表达式（因为匹配必须以「<」开头，在其他位置进行匹配尝试是徒劳的）。

使用简单预搜索可以比按部就班应用整个正则表达式快得多，原因就在于这种优化。尤其是，需要搜索的字符在目标字符串中出现的次数越少，优化越明显。同样，正则引擎判断第一个字符匹配失败的工作量越大，优化越明显。这种优化对「<i>|</i>||」比对「<(\w+)」更明显，因为在进行下一轮尝试之前，正则引擎会尝试搜索 4 个不同的多选分支，这样可以减少许多工作量。

使用模式修饰符 S 进一步优化

preg 引擎足够聪明，能把这种优化应用到大多数正则表达式，它们的匹配必须以某个字符开头，就像上面的例子一样。不过，模式修饰符 **S** 告诉引擎，对于可能以多个字符开头的表达式，必须首先分析正则表达式来启用这种优化。

这里有几个正则表达式的例子，其中有一些在本章中已经出现过，使用模式修饰符 **S** 的结果如下：

正则表达式	可能的开头字符
<(\w+) |&(\w+);	< &
[Rr]e:	R r
(Jan|Feb|…|Dec)\b	A D F J M N O S
(Re:\s*)? SPAM	R S
\S*,\s*	\x09 \x0A \x0C \x0D \x20,
[&<">]	& < " >
\r?\n\r?\n	\r \n

模式修饰符 S 没有用处的场合

想想在哪些情况下模式修饰符 S 没有用会很有其法：

- 开头有锚点的表达式（例如「^」和「\b」），或者一个锚点紧跟全局性多选分支。这限于当前的实现，「\b」的限制，理论上在未来的某些版本中可以去掉。
- 能够匹配空字符的表达式，例如「\s*」。
- 表达式可以从任何字符开始匹配（或者是绝大多数字符），例如「(?:[^()]++|\((?R)\))*」，请参考第 475 页的例子。这个表达式能够从除‘)’之外的任何字符开始匹配，所以预先检查一遍几乎不会去掉任何开始的位置。
- 开头字符只有一种可能的表达式，因为它们已经进行了优化。

使用建议

使用模式修饰符 S 之后，preg 引擎花在预分析上的时间并不会太长，所以如果你希望对大量的文本应用正则表达式，无妨使用它。如果你觉得有机会使用，潜在的可能就值得尝试。

扩展示例

Extended Examples

用这两个例子作为本章的结束。

用 PHP 解析 CSV

CSV Parsing with PHP

这里有一个用 PHP 解析 CSV（逗号分隔值）的程序，原来的例子在第 6 章（☞271）。这个正则表达式使用了占有优先量词（☞142），而不是固化分组括号，因为它们看起来更清晰。

首先，这是我们将要使用的正则表达式：

```
$csv_regex = '{
      \G(?:^|,)
      (?:
          # 或者是双引号字段 ...
          " # 起始双引号
            (  [^"]*+  (?: "" [^"]*+ )*+ )
          " # 结束双引号
        | # ...或者是...
          # ...非双引号/逗号文本...
          ( [^",]*+ )
      )
}x';
```

然后，我们用它来解析$csv文件中的一行：

```
/* 应用正则表达式，填充$all_matches */
preg_match_all($csv_regex, $line, $all_matches);

/* $Result 用来保存从$all_matches 收集的数据 */
$Result = array ();

/* 遍历每个成功的匹配... */
for ($i = 0; $i < count($all_matches[0]); $i++)
{
   /* 如果第 2 组捕获型括号匹配了，则直接使用 */
   if (strlen($all_matches[2][$i]) > 0)
      array_push($Result, $all_matches[2][$i]);
   else
   {
      /* 否则为引用字段，在使用前处理其中内嵌的相连双引号 */
      array_push($Result, preg_replace('/""/', '"', $all_matches[1][$i]));
   }
}

/* 现在可以使用$Result 数组*/
```

检查 tagged data 的嵌套正确性

Checking Tagged Data for Proper Nesting

这个例子有点复杂，它用到了许多有意思的知识：检查 XML（或者是 XHTML，或者任何标记的数据）是否包含孤立的或者错误匹配的标签。我的办法是检查正确匹配的 tag，非 tag 文本，以及自封闭 tag（self-closing tag，例如
，用 XML 的语言来说就是一个“空元素 tag”），希望我能找到整个字符串。

下面是完整的正则表达式：

```
「^((?:<(\w++)[^>]*+(?<!/)>(?1)</\2>|[^<>]++|<\w[^>]*+/>)*+)$」
```

能够匹配的字符串不会包含错误匹配的 tag（稍后会给出若干告诫）。

这可能相当复杂，但是如果分解为各个部分，就可以掌握了。外层的「^(…)$」包围表达式的主体，保证在返回 success 之前匹配整个目标字符串。主体包含在一组捕获型括号之内，我们马上会看到，这组括号容许在之后递归引用“主体”。

正则表达式的主体

正则表达式的主体，就是这三个多选分支（在正则表达式中的下画线标注，以便观察），它们包含在「(?:…)*+」中，容许任意的混合都能匹配。这三个多选分支匹配的分别是：tags、非 tag 文本，以及自封闭 tag。

因为每个多选分支能够匹配的文本之间是没有冲突的（也就是说，如果一个多选分支能够匹配，另两个就不能匹配），我知道稍后的回溯永远不会容许另一个多选分支匹配同样的文本。利用这一点，我们可以使用占有优先的星号，提高“容许任何混合”括号的匹配效率。它告诉正则引擎，不要徒劳地回溯，如果找不到匹配，就很快出结果。

因为同样的原因，三个多选分支可以以任何顺序出现，我把最可能匹配的多选分支放在最前面（☞260）。

现在逐个看这些多选分支：

第2个多选分支 非tag文本 我从它开始讲，因为「[^<>]++」很简单。这个多选分支匹配非tag文本。在这里使用占有优先量词可能有点多此一举——外面的「(?:…)*+)」也是占有优先的，但是为了安全起见，我希望在我知道不会带来负面影响的地方使用占有优先量词。（通常使用占有优先量词是为了提高效率，但是它也会改变匹配的语意。这种修改可能有帮助，不过你必须清楚它的后果☞259）。

第3个多选分支 自封闭tag 第3个多选分支「<\w[^>]*+/>」匹配自封闭tag，例如
和<img…/>（自封闭tag在后面的尖括号之前有反斜线）。与之前一样，占有优先量词可能有点多余，但它肯定不会带来负面影响。

第1个多选分支 一对匹配的tags。最后我们来看第1个多选分支：「<(\w++)[^>]*+(?<!/)>(?1)</\2>」

这个子表达式的第一部分（以下画线标注）匹配开头的tag，用「(\w++)」，也就是整个正则表达式的第2组捕获型括号（在「\w++」中使用占有优先量词是很重要的，我们将会看到）匹配tag名称。

「(?<!/)」是否定型逆序环视（☞133），确保没有匹配斜线。我们把它放在匹配开头tag的子表达式中的「>」之前，确保没有匹配自封闭tag，例如<hr/>（我们已经看到，自封闭的tag由第3个多选分支处理）。

在开头tag匹配之后，「(?1)」会递归地应用到第一组捕获型括号内的子表达式。它是之前提到的“主体”，也就是一块只包含对称tag的文本。它匹配之后应该匹配对应的结尾tag(closing tag)，就是这个多选分支的第一部分匹配的（tag的名字捕获到第二组捕获型括号）。「</\2>」开头的「</」确保它是一个结尾tag，「\2>」中的反向引用确保是一个正确的结尾tag。

如果是检查 HTML 或者其他 tag 名不区分大小写的数据，请在正则表达式之前添加「(?i)」，或者使用模式修饰符 **i**。

完成了！

占有优先量词

关于第 1 个多选分支「**<**(\w++)[^>]*+(?<!/)**>**」中的「\w**++**」的占有优先，我希望多说几句。如果流派的功能不够强大，不能使用占有优先量词或者固化分组（☞139），我会在这个多选分支的(\w+)之后加上**\b**：「<(\w+)**\b**[^>]*(?<!/)>」。

\b 很重要，它能够停止(\w+)的匹配，例如，'<**li**nk>…</**li**>'中第一个'li'的匹配。这样会将'nk'单独留在捕获型括号外面，导致后面的反向引用「\2」引用的 tag 名不完整。

正常情况下这些都不会发生，因为\w+是匹配优先的，会匹配整个 tag 名。不过，如果正则表达式应用到嵌套结构糟糕的文本中，它应该匹配失败，搜索中的回溯会强迫「\w+」匹配不完整的 tag 名，例如'<**li**nk>…</**li**>'。「\b」能解决这个问题。

谢天谢地，PHP 的强大的 preg 引擎支持占有优先量词，使用「(\w++)」与附加「\b」的意义一样：不容许回溯切割 tag 名，但是效率更高。

真实世界的 XML

真实世界的 XML 比简单的匹配 tag 要复杂得多。我们还必须考虑 XML 注释、CDATA 部分、处理指令和其他。

添加对 XML 注释的支持是很容易的，只需要增加第 4 个多选分支，「<!--.*?-->」，请务必使用「(?s)」或者是模式修饰符 **S**，这样点号能够匹配换行符。

同样，CDATA 部分的格式是<![CDATA[…]]>，可以用另一个多选分支「<!\[CDATA\[.*?]]>」来处理，'<?xml·version="1.0"?>'之类的处理指令需要再添加一个多选分支：「<\?.*?\?>」。

entity 声明的形式是<!ENTITY…>，可以用「<!ENTITY\b.*?>」来处理。XML 中有许多类似的结构，他们中的大部分可以用「<![A-Z].*?>」取代「<!ENTITY\b.*?>」来处理。

虽然还有些问题，不过上面的办法应该能够应付绝大多数XML。下面是完整的PHP代码：

```
$xml_regex = '{
   ^(
     (?: <(\w++) [^>]*+ (?<!/)> (?1) </\2>         # 匹配一组 tag
       | [^<>]++                                   # 非 tag 字符
       | <\w[^>]*+/>                               # 自封闭 tag
       | <!--.*?-->                                # 注视
       | <!\[CDATA\[.*?]]>                         # cdata 数据
       | <\?.*?\?>                                 # 处理指令
       | <![A-Z].*?>                               # Entity 声明之类
     )*+
   )$
}sx';

if (preg_match($xml_regex, $xml_string))
     echo "block structure seems valid\n";
else
     echo "block structure seems invalid\n";
```

HTML

常见的情况是，真实世界的HTML有各种各样的问题，这样的检测几乎没有实用价值，例如孤立元素或者失配的tag，以及独立出现的‘<’和‘>’字符。不过，即使是正确配对的HTML也有些特殊情况我们必须处理，注释和`<script>` tag。

HTML注释规范与XML注释一样：「`<!--.*?-->`」，使用模式修饰符s。

`<script>`部分是重要的，因为它可能包含‘<’和‘>’，所以必须容许`<script…>`和`</script>`之间出现任何字符。我们可以这样处理：「`<script\b[^>]*> .*? </script>`」。有趣的是，不包含禁止出现的‘<’和‘>’的字符的script序列会被第1个多选分支捕获，因为它走的也是“匹配的一组tag”的套路。如果`<script>`不包含任何其他字符，第1个多选分支会失败，这些文本留给新增的多选分支。

这里是HTML版本的PHP程序：

```
$html_regex = '{
   ^(
     (?: <(\w++) [^>]*+ (?<!/)> (?1) </\2>         # 匹配一对 tag
       | [^<>]++                                   # 非 tag 文本
       | <\w[^>]*+/>                               # 自封闭 tag
       | <!--.*?-->                                # 注释
       | <script\b[^>]*>.*?</script>               # script 内容
     )*+
   )$
}isx';

if (preg_match($html_regex, $html_string))
     echo "block structure seems valid\n";

else
     echo "block structure seems invalid\n";
```

索引

Index



计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书**+**视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料**(**电子书**+**视频教程**)**下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具**+**电子书**+**视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集**(**电子书**+**视频教程**) Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料**<**电子书**+**视频**>**分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍**+**视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

作者简介

Jeffrey E.F.Friedl 生长于俄亥俄州 Rootstown 的乡村，小时候希望成为天文学家，直到有一天他发现了闲置在化学实验室角落里的 TRS-80 Model I（装备了整整 16KB RAM）。1980 年他终于开始使用 Unix（和正则表达式）。在肯特（Kent）大学和新罕布什尔（New Hampshire）大学分别获得计算机学士和硕士学位之后，他在日本京都工作了 8 年，为欧姆龙公司（Omron Corporation）进行核心开发，1997 年迁居硅谷，在当时还不为人知的 Yahoo!用正则表达式处理财经新闻和数据。2004 年 4 月他偕妻儿返回京都。

Friedl 的闲暇时间很充裕，这时候他喜欢与妻子 Fumie 和 3 岁大的，总是蹦蹦跳跳的儿子 Anthony 一起。他还喜欢拍摄遍布京都的美景，照片贴在他的 blog 上：*http://regex.info/blog*。

封面介绍

《精通正则表达式（第 3 版）》的封面动物是猫头鹰（owls）。这些捕猎者包括两科（family），约 180 种，分布于世界各地，当然南极洲除外。多数种类都在夜间出动，捕食活物，小到昆虫，大到野兔。

猫头鹰的眼睛很大，直视前方，因为不易转动，猫头鹰必须转动头部才能观察四周。它们的头部最多可以旋转 270 度，有些种类的猫头鹰甚至能转到头顶向下。作为捕猎者，猫头鹰在进化过程中锻炼出极强的辨识声音频率和方向的能力。许多种类的猫头鹰的耳朵是不对称的，这样在微光或暗夜时更容易定位猎物。一旦确定了方位，它们可以依靠柔软的羽毛，出其不意，悄无声息地接近猎物。

长期以来人们将猫头鹰视为邪恶而冷血的生物，其实它们完全不是如此。或许是因为大眼睛让人觉得狡黠，民间传说中一直有它们的故事。

封面的图像来自 Dover Pictorial Archive 的 19 世纪雕版画。

Programming

O'REILLY®

精通正则表达式



本书讲解正则表达式，这种工具能够提高工作效率、让生活变得更轻松。精心调校后的正则表达式只需要十多秒就能完成以前数小时才能完成的枯燥任务。如今，正则表达式已经成为众多语言及工具——Perl、PHP、Java、Python、Ruby、MySQL、VB.NET和C#（以及.NET Framework中的任何语言）——中的标准特性，依靠它，你能以之前完全不敢设想的方式进行复杂而精巧的文本处理。

《精通正则表达式（第3版）》包含了对PHP及其正则表达式的讲解。这一版的更新也反映了其他语言的发展，深入讲解了Sun的java.util.regex，并特别提到了Java 1.4.2和Java 1.5/1.6之间的众多差异。

本书的内容：

- 各种语言和工具的功能比较
- 正则引擎的工作原理
- 优化（能节省大量的时间）
- 准确匹配期望的文本
- 针对具体语言的章节

《精通正则表达式（第3版）》，以明晰轻松的笔调向程序员深入浅出地讲解复杂的知识，并给出了现实世界中复杂问题的解决办法，读者能够立刻运用书中丰富的知识，巧妙而高效地解决各种问题。

"如果你的工作需要用到正则表达式（即便你已经有本很不错的关于开发语言的书），我还是要向你强烈推荐本书。"

——Dr.Chris Brown, Linux Format

"毫不夸张地说，《精通正则表达式（第3版）》是学习该工具的不二选择，也是每个程序员必备的杰作。"

——Jason Menard, Java Ranch

"所有关于正则表达式的书中，找不到比这更好的了。"

——Zak Greant, Planet PHP

图书分类：程序设计

责任编辑：周筠 王继花
项目管理：梁晶



www.phei.com.cn

网上订购：www.dearbook.com.cn
第二书店·第一服务





ISBN 978-7-121-04684-1

定价：75.00元

O'Reilly Media, Inc.授权电子工业出版社出版